Programmierkurs Java

Dr.-Ing. Dietrich Boles

Aufgaben zu UE 20 – Interfaces

(Stand 12.04.2021)

Aufgabe 1:

Stellen Sie sich folgendes Szenario vor: Eine Person ist immer an wichtigen Ereignissen interessiert. Deshalb implementiert sie ein Interface `NachrichtenEmpfaenger`:

```
interface NachrichtenEmpfaenger {
     // Uebergabe einer neuen Nachricht
     public void empfangeNachricht(String nachricht);
}
```

Stellen Sie sich vor, dass ein Nachrichten-Empfänger eine empfangene Nachricht im einfachsten Fall auf den Bildschirm ausgibt.

Nachrichten können wiederum von verschiedenen Quellen erzeugt werden, z.B. Radio, TV, Zeitung. Die Fähigkeit, Nachrichten zu versenden, lässt sich somit auch in ein Interface `NachrichtenQuelle` abstrahieren:

```
interface NachrichtenQuelle {

     // Interessierte können sich bei der Quelle anmelden
     // (falls sie noch nicht angemeldet sind)
     public void anmelden(NachrichtenEmpfaenger empf);

     // Angemeldete können sich bei der Quelle wieder abmelden
     // (falls sie angemeldet sind)
     public void abmelden(NachrichtenEmpfaenger empf);

     // neue Nachricht wird an alle angemeldeten
     // Empfaenger uebergeben
     // (Aufruf deren Methode empfangeNachricht)
     public void sendeNachricht(String nachricht);
}
```

Etwas umständlich an dieser Struktur ist, dass sich ein Nachrichten-Empfänger bei jeder Nachrichten-Quelle anmelden muss, von der er neue Nachrichten zugeschickt bekommen möchte. Günstiger wäre ein Vermittler, der sich bei mehreren Nachrichten-Quellen anmeldet und alle Nachrichten, die er von den Quellen bekommt, direkt an Nachrichten-Empfänger weiterleitet, die sich bei ihm angemeldet haben. Ein Vermittler ist damit sowohl Nachrichten-Empfänger als auch Nachrichten-Quelle.

Aufgabe: Definieren und implementieren Sie eine solche Klasse `Vermittler`, die die beiden Interfaces `NachrichtenEmpfaenger` und `NachrichtenQuelle` entsprechend implementiert.

Aufgabe 2:

Schauen Sie sich das folgende Interface sowie die beiden folgenden Klasse an:

```
interface Vergleichbar {
     /*
      * vergl. das aufgerufene Objekt mit dem als Parameter uebergebenen
      * Objekt; liefert: -1 falls das aufgeruf. Objekt kleiner ist als das
      * Paramobjekt, 0 falls beide Objekte gleich gross sind, 1 falls das
      * aufgerufene Objekt groesser ist als das Parameterobjekt
      */
     public int vergleichenMit(Vergleichbar obj);
}

class NuetzlicheFunktionen {

     // liefert ein "kleinstes" (auf der Basis der
     // Vergleichbar-Implementierung!) Element des Parameter-Arrays
     // Achtung: Man kann davon ausgehen, dass das Parameter-Array
     // mindestens 1 Element enthaelt
     public static Vergleichbar kleinstesElement(Vergleichbar[] elemente)
{
          // todo
     }
}

class Integer {
     protected int wert;

     public Integer(int w) {
          this.wert = w;
     }

     public int getWert() {
          return this.wert;
     }
}
```

Aufgabe:

(1) Implementieren Sie die Methode *kleinstesElement* der Klasse *NuetzlicheFunktionen*

(2) Leiten Sie von der Klasse *Integer* eine Klasse *VInteger* ab, die das Interface *Vergleichbar* implementiert

(3) Schreiben Sie ein kleines Testprogramm, das zunächst ein Array mit VInteger-Objekten erzeugt und initialisiert, anschließend die Funktion *kleinstesElement* mit diesem Array aufruft und den Wert des ermittelten kleinsten Elements auf den Bildschirm ausgibt

Aufgabe 3:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können. Schauen Sie sich dazu das folgende Interface an:

```
interface Chiffrierung {
     public char chiffrieren(char zeichen);
     public char dechiffrieren(char zeichen);
}
```

Das Chiffrieren ist ein Verschlüsselungsverfahren, bei dem jeder Buchstabe in einem Text durch einen anderen Buchstaben ersetzt wird. Die Methode `chiffrieren` des Interfaces soll eine entsprechende Umsetzung des übergebenen Zeichen vornehmen. Die Methode `dechiffrieren` bildet die reverse Funktion.

Ihre Aufgabe besteht nun darin, das Interface zweimal zu implementieren:

- Bei der ersten Implementierung sollen Sie die so genannte *Caesar-Verschiebung* implementieren. Die Caesar-Verschiebung beruht auf einem Geheimtextalphabet, das um eine bestimmte Stellenzahl *n* gegenüber dem Klartextalphabet verschoben ist. Beispiel für n = 3: a -> d, b -> e, c -> f, ..., w -> z, x -> a, y -> b, z -> c. Chiffriert werden sollen hier nur Kleinbuchstaben. Alle anderen Zeichen sollen unverändert zurückgegeben werden. Implementieren Sie neben den beiden Methoden des Interfaces einen Konstruktor, dem die Stellenzahl als Parameter übergeben wird
- Bei der zweiten Implementierung des Interface übergeben Sie im Konstruktor explizit das Geheimtextalphabet als 26elementiges char-Array *m*. Die Chiffrierung erfolgt hierbei durch: a -> m[0], b -> m[1], ..., z -> m[25]. Chiffriert werden sollen auch hier nur Kleinbuchstaben. Alle anderen Zeichen sollen unverändert zurückgegeben werden.

Implementieren Sie anschließend die folgende Klasse:

```
public class Verschluesselung {
  public static String verschluesseln(String klartext,
                                      Chiffrierung schluessel);
  public static String entschluesseln(String geheimtext,
                                      Chiffrierung schluessel);
}
```

zum Ver- bzw. Entschlüsseln von Botschaften gemäß eines zwischen Sender und Empfänger vereinbarten Chiffrierungsschlüssels.

Implementieren Sie weiterhin ein Testprogramm (= Aufruf aller Methoden) für die Klasse *Verschluesselung*, in dem beide Chiffrierungsverfahren eingesetzt werden

Aufgabe 4:

Sie haben in Aufgabe 3 mit der Caesarverschlüsselung ein Verschlüsselungsverfahren kennen gelernt, bei dem jeder Buchstabe der Klartextes im Geheimtext durch einen anderen Buchstaben ersetzt wird. Dieses

Verschlüsselungsverfahren ist recht einfach zu knacken, und zwar mit Hilfe einer Häufigkeitsanalyse. Diese beruht darauf, dass in Texten einige Buchstaben häufiger vorkommen als andere. Man braucht also nur die Häufigkeit des Auftretens eines Buchstaben im Geheimtext zu ermitteln und kann dann daraus Schlussfolgerungen ziehen, welcher tatsächliche Buchstabe sich hinter dem Geheimtextbuchstaben verbirgt.

In dieser Aufgabe sollen Sie ein Java-Programm entwickeln, das eine Häufigkeitsanalyse durchführt. Im Detail soll das Programm folgenden Algorithmus implementieren:

- Zunächst wird ein Benutzer aufgefordert, einen String einzugeben.
- Anschließend erstellt das Programm eine Tabelle mit der prozentualen Häufigkeit des Auftretens einzelner Buchstaben (nur die Kleinbuchstaben!!!) im eingelesenen String.
- Im Anschluss daran kann ein Nutzer in einer Schleife die prozentuale Häufigkeit des Vorkommens einzelner Kleinbuchstaben im eingelesenen String abfragen.

Beispiel für ein typisches Szenario (Benutzereingaben stehen in <>):

```
String eingeben: <Dies ist ja nur wirklich eine einfache Aufgabe !>
Prozentuale Häufigkeit von: <e>
Prozentuale Häufigkeit des Buchstabens e im String = 12.76 %
Prozentuale Häufigkeit von: <b>
Prozentuale Häufigkeit des Buchstabens b im String = 2.12 %
Prozentuale Häufigkeit von: <A>
A ist kein Kleinbuchstabe!
Prozentuale Häufigkeit von: <x>
Prozentuale Häufigkeit des Buchstabens x im String = 0.0 %
.....
```

Aufgabe 5:

In dieser Aufgabe geht es wieder um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Nachdem Sie in Aufgabe 3 schon zwei Verschlüsselungsverfahren kennen gelernt haben, sollen Sie nun ein drittes implementieren, und zwar die Vignere-Verschlüsselung.

Die Vignere-Verschlüsselung basiert auf dem so genannten Vignere-Quadrat:

```
Klar   a b c d e f g h i j k l m n o p q r s t u v w x y z

0      a b c d e f g h i j k l m n o p q r s t u v w x y z
1      b c d e f g h i j k l m n o p q r s t u v w x y z a
2      c d e f g h i j k l m n o p q r s t u v w x y z a b
3      d e f g h i j k l m n o p q r s t u v w x y z a b c
…
11     l m n o p q r s t u v w x y z a b c d e f g h i j k
```

```
…
24      y z a b c d e f g h i j k l m n o p q r s t u v w x
25      z a b c d e f g h i j k l m n o p q r s t u v w x y
```

Zeile 1 des Quadrates enthält ein Geheimtextalphabet mit einer Caesar-Verschiebung von 1; Zeile 2 des Quadrates enthält ein Geheimtextalphabet mit einer Caesar-Verschiebung von 2; usw.

Vorteil der Vignere-Verschlüsselung ist, dass eine Häufigkeitsanalyse zum Knacken des Geheimcodes versagt (siehe auch Aufgabe 4). Bei der Vignere-Verschlüsselung geht man nämlich so vor, dass man jeden Buchstaben einer geheim zu haltenden Botschaft anhand einer anderen Zeile des Vignere-Quadrates verschlüsselt. Um die Botschaft wieder entschlüsseln zu können, muss der Empfänger allerdings wissen, welche Zeile des Vignere-Quadrates für den jeweiligen Buchstaben benutzt wurde. Deshalb tauschen Sender und Empfänger vorher ein Schlüsselwort aus.

Nehmen wir einmal an, die zu verschlüsselnde Botschaft sei „truppenabzugnachosten“ und das Schlüsselwort sei „licht“. Zunächst wird das Schlüsselwort über die Botschaft geschrieben und so lange wiederholt, bis jeder Buchstabe der Botschaft mit einem Buchstaben des Schlüsselwortes verknüpft ist.

```
Schlüsselwort     lichtlichtlichtlichtl
Klartext          truppenabzugnachosten
Geheimtext        ezwwipvcisfoehvswuaxy
```

Der Geheimtext wird dann folgendermaßen erzeugt: Um den ersten Buchstaben „t“ zu verschlüsseln, stellen wir zunächst fest, dass über ihm der Buchstabe „l“ steht, der wiederum auf eine bestimmte Zeile des Vignere-Quadrates verweist. Die mit „l“ beginnenden Reihe 11 enthält das Geheimtextalphabet, das wir benutzen, um den Stellvertreter des Klarbuchstabens „t“ zu finden. Also folgen wir der Spalte unter „t“ bis zum Schnittpunkt mit der Zeile „l“, und dort befindet sich der Buchstabe „e“ im Geheimtext. Genauso gehen wir mit allen weiteren Buchstaben der Botschaft vor.

Die Entschlüsselung eines Zeichen eines Geheimtextes erfolgt auf umgekehrten Weg: Anhand des aktuellen Schlüsselzeichens wird die aktuelle Zeile des Quadrates bestimmt. Aus der Spalte des Zeichens leitet sich dann das Klartextzeichen ab.

Konkrete Aufgabe: Implementieren Sie eine Klasse Vignere, die das folgende Interface Verschluesselung gemäß der Vignere-Verschlüsselung implementiert. Verschlüsselt werden sollen nur Kleinbuchstaben. Alle anderen Buchstaben sollen unverändert bleiben

```
interface Verschluesselung {
  public String verschluesseln(String klartext);
  public String entschluesseln(String geheimtext);
}
```

Das folgende Gerüst sei dabei vorgegeben:

```
public class Vignere implements Verschluesselung {

    // enthaelt nur Kleinbuchstaben!
    private String schluesselwort;

    // speichert das Vignere-Quadrat
    private char[][] quadrat = null;
```

```
	// schluessel darf nur Kleinbuchstaben enthalten!
	public Vignere(String schluessel) {
		this.schluesselwort = schluessel;
		// initialisieren des Vignere-Quadrates
		this.quadrat = new char[26][26];
		for (int i = 0; i < 26; i++) {
			for (int j = 0; j < 26; j++) {
				this.quadrat[i][j] = (char) ('a' + (i + j) % 26);
			}
		}
	}

	// liefert zu einem uebergebenen Klartext unter Nutzung des im
	// Attribut schluesselwort gespeicherten Schluesselwortes den
	// entsprechenden Geheimtext
	public String verschluesseln(String klartext) {

		// muss von Ihnen implementiert werden!

	}

	// liefert zu einem uebergebenen Geheimtext unter Nutzung des
	// im Attribut schluesselwort gespeicherten Schluesselwortes
	// den entsprechenden Klartext
	public String entschluesseln(String geheimtext) {

	    // muss von Ihnen implementiert werden!

	}
}
```

Aufgabe 6:

Implementieren Sie eine Methode *check*, die (auf eine von Ihnen zu konzipierende Art und Weise) als ersten Parameter eine beliebige mathematische Funktion f mit dem Definitionsbereich int und dem Wertebereich int, und als zweiten und dritten Parameter die Unter- und Obergrenze eines geschlossenen int-Intervals übergeben bekommt. Die Methode *check* soll überprüfen (und einen entsprechenden booleschen Wert liefern), ob die übergebene Funktion f im übergebenen Interval linear ist (genauer: f(x) = n*x für alle x im Interval, und n ist ein beliebiger int-Wert zwischen 1 und 100)).

Implementieren Sie weiterhin ein Testprogramm, in dem die Methode *check* getestet wird!

Aufgabe 7:

In dieser Aufgabe geht es darum, die Größe der Oberflächen zweier volumenmäßig ungefähr gleich großer 3-dimensionaler Behälterobjekte miteinander zu vergleichen. Schauen Sie sich dazu die folgenden Interfaces/Klassen an:

```
interface Behaelter {
	public double getOberflaeche();

	public double getVolumen();

	public void veraendern(double inkrement);
```

```
}

class Wuerfel implements Behaelter {
	private double kantenLaenge;

	public Wuerfel(double kantenLaenge) {
		this.kantenLaenge = kantenLaenge;
	}

	public double getOberflaeche() {
		return 6.0 * this.kantenLaenge * this.kantenLaenge;
	}

	public double getVolumen() {
		return	this.kantenLaenge	*	this.kantenLaenge	*
this.kantenLaenge;
	}

	public void veraendern(double inkrement) {
		this.kantenLaenge += inkrement;
	}
}

class Kugel implements Behaelter {
	static final double PI = 3.1415;

	private double radius;

	public Kugel(double radius) {
		this.radius = radius;
	}

	public double getOberflaeche() {
		return 4.0 * PI * this.radius * this.radius;
	}

	public double getVolumen() {
		return	4.0	/	3.0	*	PI	*	this.radius	*	this.radius	*
this.radius;
	}

	public void veraendern(double inkrement) {
		this.radius += inkrement;
	}
}
```

Aufgabe: Schreiben Sie durch Benutzung dieser Klassen (sowie der bekannten Klasse IO) ein Java-Programm, das folgendes tut:

- Zunächst werden ein Wuerfel- und ein Kugel-Objekt mit jeweils einer Größe (Kantenlänge bzw. Radius) erzeugt, die zuvor vom Benutzer abgefragt wird. Achten Sie darauf, dass der Nutzer "ordentliche" Werte eingibt.
- Anschließend wird das Volumen der beiden Objekte auf den Bildschirm ausgegeben
- Danach wird ein Inkrementwert inkrement vom Typ double vom Benutzer abgefragt.
- Anschließend werden in einer Schleife die Volumen der beiden Objekte "angeglichen", und zwar auf folgende Art und Weise: Das anfangs volumenmäßig größere Objekt wird in jedem Schleifendurchlauf um den

Inkrementwert verkleinert (Methode veraendern) und das anfangs volumenmäßig kleinere Objekt wird um den Inkrementwert vergrößert. Die Schleife wird solange durchlaufen, bis das anfangs volumenmäßig größere Objekt volumenmäßig kleiner ist als das anfangs volumenmäßig kleinere Objekt.

- Zum Schluss werden Volumen und Oberfläche der beiden Objekte auf den Bildschirm ausgegeben.

Im Folgenden wird ein Beispiel für eine mögliche Ausgabe des Programms gegeben (in <> die Eingaben des Benutzers):

```
Kantenlaenge: <10.0>
Radius: <9.0>
Wuerfel-Volumen (Anfang): 1000.0
Kugel-Volumen (Anfang): 3053.5
Inkrement: <0.001>
Wuerfel-Volumen: 1612.3
Wuerfel-Oberfläche: 824.9
Kugel-Volumen: 1612.1
Kugel-Oberflaeche: 664.8
```

Aufgabe 8:

Schreiben Sie ein Programm, das folgende GUI erzeugt:

Im linken Teil befindet sich ein Label (java.awt.Label) mit dem Text „Eingabe“, im rechten Teil ein TextField (java.awt.TextField), in dem der Nutzer einen Text eingeben kann.

Immer wenn ein Nutzer einen Text eingegeben und mit <RETURN> abgeschlossen hat, soll die entsprechende Eingabe auf der Console ausgegeben werden. Weiterhin soll im TextField der eingegebene Text gelöscht werden.

Die Behandlung von Nutzereingaben erfolgt analog zu dem in der Unterrichtseinheit 20 behandelten Verfahren zur Registrierung von Button-Klicks.

Aufgabe 9:

Gegeben sei das folgende Java-Programm:

```
class Tierstimmen {
     public static void main(String[] args) {
           Tier[] tiere = new Tier[5];
           for (int i = 0; i < 5; i++) {
                 tiere[i] = erzeugeTier();
           }
```

```
            for (int i = 0; i < 5; i++) {
                  tiere[i].gibLaut();
            }
      }

      public static Tier erzeugeTier() {
            String eingabe = IO.readString("Tier (Hund/Katze): ");
            if (eingabe.equals("Hund")) {
                  return new Hund();
            } else {
                  return new Katze();
            }
      }
}
```

Implementieren Sie geeignete Interfaces bzw. Klassen `Hund` und `Katze`, damit sich das Programm compilieren lässt und für Katzen "Miau" und für Hunde "Wau wau!" auf den Bildschirm ausgibt.

Aufgabe 10:

Schauen Sie sich das folgende Java-Programm an:

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Frame;
import java.awt.Label;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;

/*
 aus dem AWT:

 public interface ActionListener {
     public void actionPerformed(ActionEvent e);
 }
 */

public class UE34Aufgabe10 {

      public static void main(String[] args) {
            MyFrame f = new MyFrame();
            f.setBounds(200, 200, 100, 80);
            f.setVisible(true);
            // ab hier ist der Event-Manager des AWT aktiv
      }
}

interface StateChangeListener {
      public void stateHasChanged();
}

final class MyFrame extends Frame implements ActionListener,
            StateChangeListener {

      private Button button;

      private Label label;
```

```
	private int numberOfClicks;

	private Vector<StateChangeListener> listeners;

	public MyFrame() {
		super("MyFrame");
		this.numberOfClicks = 0;
		this.listeners = new Vector<StateChangeListener>();
		this.button = new Button("Please click");
		this.label = new Label("" + this.numberOfClicks);

		// fuegt Button und Label in das Fenster ein
		this.add(this.button, BorderLayout.NORTH);
		this.add(this.label, BorderLayout.SOUTH);

		// registriert Listener
		this.addStateChangeListener(this);
		this.button.addActionListener(this);
	}

	// wird aufgerufen, wenn der Button geklickt wird
	public void actionPerformed(ActionEvent e) {
		this.numberOfClicks++;

		// alle Interessenten der Zustandsaenderung werden
		// informiert
		for (StateChangeListener elem : this.listeners) {
			elem.stateHasChanged();
		}
	}

	// fuegt ein Objekt ein, das Interesse daran hat,
	// ueber Zustandsaenderung der Klickanzahl informiert zu werden
	public void addStateChangeListener(StateChangeListener listener) {
		this.listeners.add(listener);
	}

	// das Fenster selbst ist auch daran interessiert,
	// ueber Zustandsaenderungen der Klickanzahl informiert zu werden;
	// falls dies passiert, wird der Text des Labels auf die aktuelle
	// Anzahl der Button-Klicks gesetzt
	public void stateHasChanged() {
		this.label.setText("" + this.numberOfClicks);
	}

	// liefert die Anzahl an Klicks auf den Button
	public int getNumberOfClicks() {
		return this.numberOfClicks;
	}
}
```

Führt man dieses Programm aus, erscheint auf dem Bildschirm ein neues Fenster mit einem Button und einem Label. Jedes Mal, wenn der Benutzer auf den Button klickt, wird die Zahl im Label um 1 erhöht. In der folgenden Abbildung hat der Benutzer fünfmal auf den Button geklickt.

Aufgabe: Erweitern Sie das Programm derart, dass jedes Mal, wenn der Benutzer auf den Button klickt, zusätzlich die aktuelle Klickanzahl auch auf der Standard-Ausgabe (System.out) ausgegeben wird (siehe folgende Abbildung).

Aber **Achtung**: Sie dürfen die Klasse `MyFrame` nicht verändern! Implementieren Sie stattdessen einen neuen *StateChangeListener* und registrieren diesen in der main-Prozedur beim Fenster.

Aufgabe 11:

Gegeben sei folgender Ausschnitt aus einem Java-Programm:

```
class Vogel {
     public void fliegen() {
     }
}

class Katze {
     public void laufen() {
     }
}
```

```
class Fisch {
     public void schwimmen() {
     }
}

class Tierwelt {

     public static void bewegen(Object object) {
           if (object.getClass().getName().equals("Vogel"))
                 ((Vogel) object).fliegen();
           else if (object.getClass().getName().equals("Katze"))
                 ((Katze) object).laufen();
           else if (object.getClass().getName().equals("Fisch"))
                 ((Fisch) object).schwimmen();
     }
}
```

Zum Hintergrund des Programms: Ein Programmierer A möchte anderen Programmierern in einer Klasse *Tierwelt* Funktionen zur Verfügung stellen, die bestimmte Methoden von Tier-Klassen aufrufen. Im konkreten Beispiel implementiert er eine Funktion *bewegen*, die die entsprechende "Bewegungsmethode" des übergebenen Objektes aufruft.

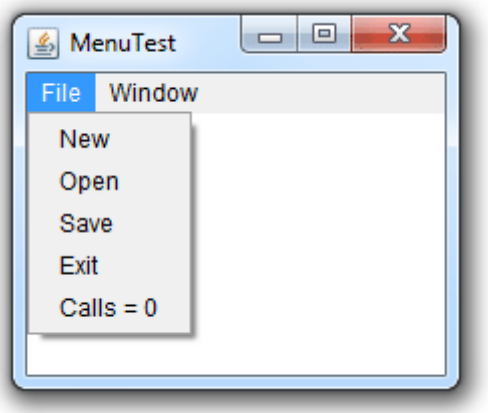
Es gibt aber ein Problem: Immer wenn ein Programmierer eine weitere Tier-Klasse hinzufügt, muss Programmierer A den Source-Code der Funktion *bewegen* ändern.

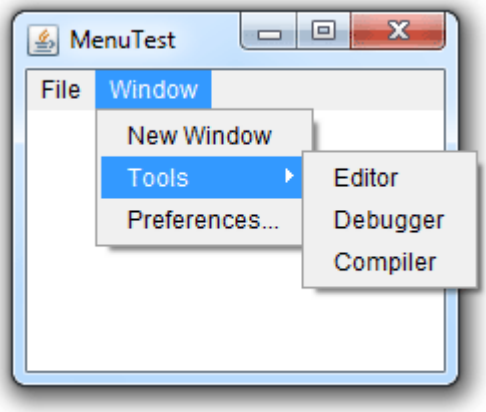
Aufgabe: Helfen Sie Programmierer A! Schreiben Sie die Funktion *bewegen* und evtl. auch andere Teile des obigen Programms so um, dass die Funktion *bewegen* immer die korrekte "Bewegungsmethode" einer Tier-Klasse aufruft, und zwar ohne dass ihr Source-Code geändert werden muss, wenn eine neue Tier-Klasse hinzukommt! Nutzen Sie dazu die Konzepte der Polymorphie und des dynamischen Bindens.

Achtung: Sie dürfen in Ihrem Programm keine if-, switch-, while-, for- und do-Anweisungen verwenden!

Aufgabe 12:

In dieser Aufgabe geht es um die Erstellung einer GUI:



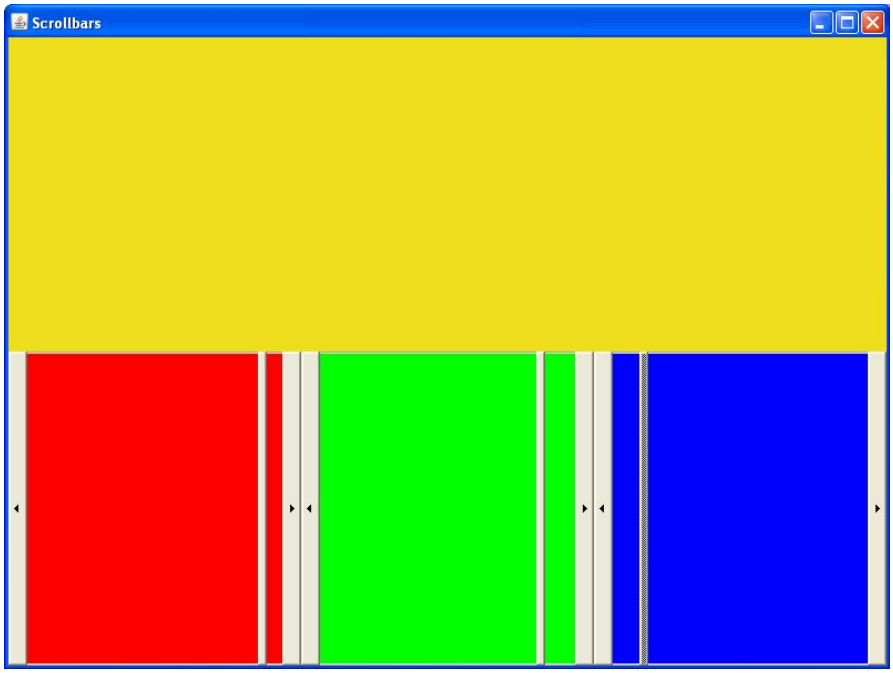


Aufgabe (a): Schreiben Sie ein kleines Programm, das ein Fenster mit den oben dargestellten Menüs erzeugt. Schauen Sie sich dazu die Klassen `java.awt.Frame`, `java.awt.MenuBar`, `java.awt.Menu` und `java.awt.MenuItem` an. Von besonderer Bedeutung sind die Methoden `setMenuBar` der Klasse `Frame` und die Methode `add` der Klasse `MenuBar` und der Klasse `Menu`.

Aufgabe (b): Realisieren Sie, dass beim Auswählen des letzten Menü-Items des File-Menüs der Label geändert wird. Und zwar soll dort immer die Anzahl der Aufrufe dieses Items angezeigt werden („Calls = 1“, Calls = 2“, …). Hinweis: Das Vorgehen zum Zuordnen einer Aktion zu einem MenuItem ist identisch mit dem Zuordnen einer Aktion zu einem Button.

Aufgabe 13:

Entwickeln Sie ein Java-Programm mit einer interaktiven GUI, bei der ein Benutzer mit Hilfe dreier Scrollbars die Farbe eines Panels beeinflussen kann.



Im oberen Bereich des Fensters (Klasse `java.awt.Frame`) befindet sich ein Panel (Klasse `java.awt.Panel`), darunter befinden sich drei Scrollbars (Klasse `java.awt.Scrollbar`). Anfangs ist das Panel schwarz und die Regler der drei Scrollbars befinden sich jeweils ganz links. Sie sollen nun folgende Benutzereingriffe implementieren:

Die Farbe des Panels soll sich ergeben aus dem Stand der Regler der drei Scrollbars. Verschiebt der Benutzer den Regler des linken Scrollbars nach rechts, wird der Rot-Wert des Panels erhöht, verschiebt er den Regler nach links, wird der Rot-Wert erniedrigt. Entsprechend der RGB-Farben bewegt sich der Wert immer zwischen 0 (Regler links, kein Rot) und 255 (Regler rechts, volles Rot). Der mittlere Scrollbar regelt analog den Farbanteil von grün, der rechte den Farbanteil von blau.

Hinweise:

- Schauen Sie sich in der Java-API-Dokumentation die Klassen `java.awt.Panel` und `java.awt.Scrollbar` an.
- Farben lassen sich in Java durch die Klasse `java.awt.Color` repräsentieren.
- Die Farbe eines Panels sowie die Farbe von Scrollbars werden jeweils über die Methode `setBackground` festgelegt.
- Die Registrierung und Behandlung von Bewegungen eines Scrollbar-Reglers erfolgt durch so genannte `AdjustmentListener`. Diese müssen einem Scrollbar über die Methode `addAdjustmentListener` zugeordnet werden. Das Prinzip ist analog zum Prinzip der `ActionListener`, die wir in den Vorlesungen für Buttons und MenuItems kennen gelernt haben. Schauen Sie sich in der Java-API-Dokumentation das Interface `java.awt.event.AdjustmentListener` und die Klasse `java.awt.event.AdjustmentEvent` an.

Aufgabe 14:

Definieren Sie eine ADT-Klasse `UKWRadio` zur Repräsentation eines UKW-Radios. Der Zustand des Radios ist gegeben durch eine Frequenz zwischen 87.5 und 108.0 MHz. Nach dem Erzeugen des Radio-Objektes beträgt die Frequenz 87.5 MHz. Die Frequenz kann in den angegebenen Frequenzen schrittweise um 0.5 MHz nach oben bzw. unten verändert werden. Weiterhin stehen N (> 0) Stationstasten zur Verfügung, auf denen man Frequenzen speichern bzw. gespeicherte Frequenzen wieder abrufen kann.

Insgesamt soll Ihre Klasse also die folgenden Methoden definieren:

- Einen Konstruktor, dem die Anzahl an Stationstasten übergeben wird
- Eine Methode, die die aktuelle Frequenz liefert
- Eine Methode zum Verringern der Frequenz um 0.5 MHZ
- Eine Methode zum Erhöhen der Frequenz um 0.5 MHZ
- Eine Methode, um die aktuelle Frequenz auf einer Stationstaste zu speichern
- Eine Methode, um die aktuelle Frequenz auf die Frequenz einer angegebenen Stationstaste einzustellen

Entwickeln Sie weiterhin ein Testprogramm für die Klasse `UKWRadio` mit einer graphisch-interaktiven Oberfläche analog zur folgenden Abbildung (2 Stationstasten).

Aufgabe 15:

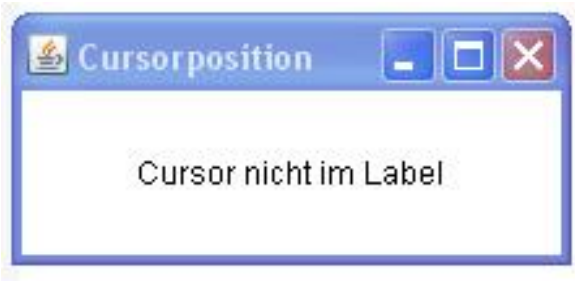
Schauen Sie sich die Klasse `java.util.Observable` und das Interface `java.util.Observer` an.

Leiten Sie zunächst von der Klasse `Observable` eine Klasse `Number` ab, in der eine Zahl (int) beobachtet wird. Immer wenn sich diese Zahl ändert, sollen Observer darüber informiert werden.

Schreiben Sie eine GUI mit einem Button und einem Label. Bei einem Button-Klick soll eine beobachtete Number erhöht werden. Die geänderte Zahl soll dann anschließend sowohl in dem Label als auch auf der Console ausgegeben werden. Realisieren Sie dies über das Interface `Observer`.

Aufgabe 16:

Entwickeln Sie ein Java-Programm, das ein Fenster auf dem Bildschirm öffnet. In dem Fenster ist als einzige Komponente ein Label (eine GUI-Komponente zur Darstellung von Strings) platziert, das anfangs den String „Cursor nicht im Label“ darstellt. Sobald der Benutzer den Mauscursor oberhalb des Labels bewegt, sollen jeweils die aktuellen Mauskoordinaten in der Form „x:y“ im Label erscheinen. Wird der Mauscursor aus dem Label entfernt, soll wieder der String „Cursor nicht im Label“ dargestellt werden.



Das Fenster ist eine Instanz der Klasse `java.awt.Frame`, die Sie aus der Veranstaltung bereits kennen. Ansonsten benötigen Sie folgende Klassen und Methoden:

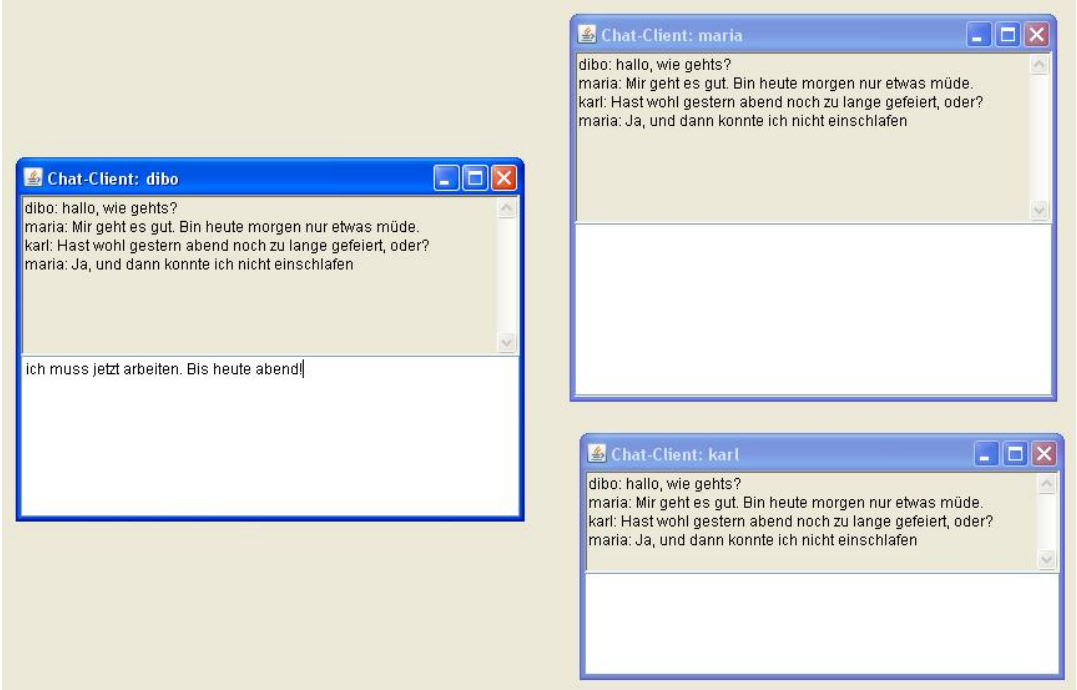
```
public class Label extends java.awt.Component {  // aus dem Paket java.awt

      // stellt den übergebenen String im Label dar
      public Label(String str)

      // ändert den im Label dargestellten String
      public void setText(String str)

      // registriert bei dem Label einen MouseListener
      public void addMouseListener(MouseListener l)

      // registriert bei dem Label einen MouseMotionListener
      public void addMouseMotionListener(MouseMotionListener l)
}


public class MouseEvent {  // aus dem Paket java.awt.event

      // liefert die aktuelle x-Koordinate des Mauscursors relativ zur betroffenen
```

```
      // Komponente
      public int getX()

      // liefert die aktuelle y-Koordinate des Mauscursors relativ zur betroffenen
      // Komponente
      public int getY()
}


public interface MouseListener {  // aus dem Paket java.awt.event

      // wird aufgerufen, wenn der Mauscorsor eine Komponente verlässt;
      // über das Event-Objekt werden weitergehende Informationen zum aktuellen
      // Status der Maus übergeben
      public void mouseExited(MouseEvent e);

      // wird aufgerufen, wenn der Mauscorsor eine Komponente betritt;
      // über das Event-Objekt werden weitergehende Informationen zum aktuellen
      // Status der Maus übergeben
      public void mouseEntered(MouseEvent e);

}

public interface MouseMotionListener { // aus dem Paket java.awt.event

      // wird aufgerufen, wenn die Maus oberhalb einer Komponente bewegt wird;
      // über das Event-Objekt werden weitergehende Informationen zum aktuellen
      // Status der Maus übergeben
      public void mouseMoved(MouseEvent e);
}
```

Der Mechanismus zum Agieren bei entsprechenden Mausoperationen erfolgt analog zum aus der Unterrichtseinheit 20 bekannten Reagieren auf Mausklicks auf Buttons.

Aufgabe 17:

Bei dieser Aufgabe sollen Sie ein einfaches Chat-System realisieren. Dieses besteht aus einem Chat-Server, bei dem sich prinzipiell beliebig viele Chat-Clients anmelden bzw. auch wieder abmelden können.



Hinweise:

- Implementieren Sie 3 Klassen: `ChatServer`, `ChatClient`, `ChatSystem`.
- Die Klasse `ChatServer` ist von der Klasse `java.util.Observable` abgeleitet. Ein Objekt der Klasse realisiert das Zusammenspiel der Clients, die sich als Observer bei ihm angemeldet haben.
- Die Klasse `ChatClient` ist ein Fenster (`java.awt.Frame`), das das Interface `java.util.Observer` implementiert. Es enthält im oberen Teil eine Anzeigekomponente (`java.awt.TextArea`) und im unteren Teil eine Eingabekomponente (`java.awt.TextField`). Gibt ein Benutzer in der Eingabekomponente einen Text ein und schließt ihn mit der <RETURN>-Taste ab, soll der Text inklusive des Benutzernamens als Präfix in den Anzeigekomponenten aller beim Chat-Server angemeldeter Chat-Clients erscheinen. Der Text der Eingabekomponente wird wieder gelöscht.
- Die Klasse `ChatSystem` ist das Hauptprogramm. Es wird mit `java ChatSystem <user1> <user2> … <userN>` aufgerufen. Es erzeugt einen Chat-Server und für alle als Parameter übergebenen Benutzernamen einen Chat-Client. Die Chat-Clients werden jeweils beim Chat-Server als Observer angemeldet.

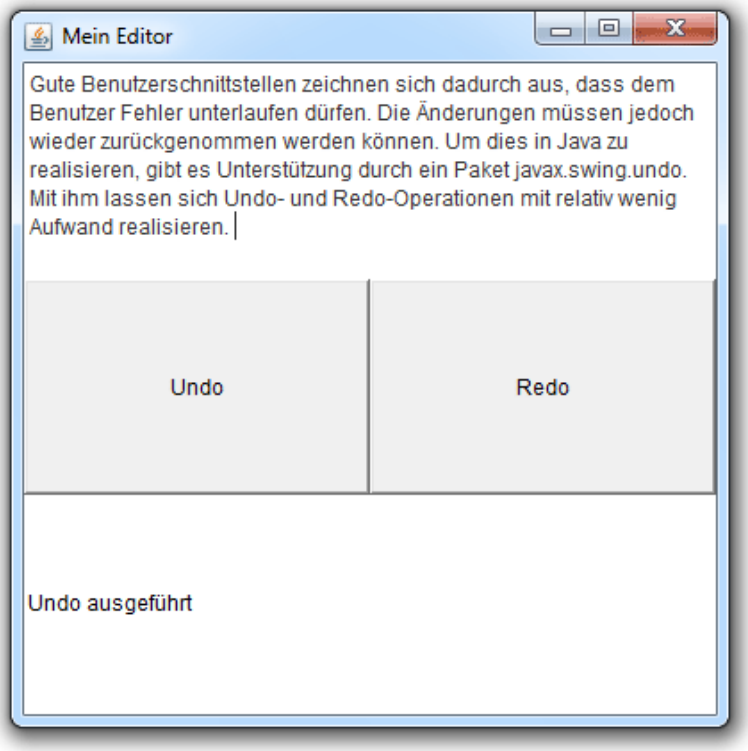
Aufgabe 18:

Bei dieser Aufgabe sollen Sie einen graphischen Editor mit Undo-Redo-Funktionalität implementieren. Konkret soll die zu erstellende GUI aus folgenden Komponenten bestehen:

- oben einer `javax.swing.JTextPane`-Komponente (das ist eine Java-Swing-Komponente mit vorgefertigter Editor-Funktionalität)
- darunter zwei `java.awt.Button`-Objekte, links für das Undo, rechts für das Redo.
- unten eine `java.awt.Label`-Komponente

In der JTextPane-Komponente kann ein Benutzer über die Tastatur Eingaben tätigen. Drückt er auf den Undo-Button soll, falls möglich, die letzte in der JTextPane-Komponente getätigte Aktion rückgängig gemacht werden und in jedem Fall in der Label-Komponente eine passende Ausgabe erscheinen. Drückt er auf den Redo-Button soll, falls möglich, die letzte Undo-Aktion rückgängig gemacht werden und in jedem Fall in der Label-Komponente eine passende Ausgabe erscheinen.

Hinweis: Im JDK steht bereits im Paket `javax.swing.undo` eine Klasse `UndoManager` zur Verfügung, die Methoden für ein Undo bzw. Redo sowie Methoden zum Überprüfen, ob die entsprechenden Aktionen überhaupt möglich sind, definiert. Schauen Sie sich diese Klasse genau an. Um ein UndoManager-Objekt mit Ihrer JTextPane-Komponente zu verknüpfen, nutzen Sie bitte die Methode `editor.getStyledDocument().addUndoableEditListener` (`editor` sei dabei eine Referenz auf Ihre JTextPane-Komponente).



Aufgabe 19:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Eine der beiden grundlegenden Verschlüsselungsklassen ist die *Transposition*. Dabei werden die Zeichen einer Botschaft (des Klartextes) umsortiert. Jedes Zeichen bleibt zwar unverändert erhalten, jedoch wird die Stelle, an der es steht, geändert. Als sehr einfaches und anschauliches Beispiel einer geregelten Transposition soll hier die *Gartenzaun-Transposition* dienen: Die Buchstaben des Klartextes werden abwechselnd auf zwei Zeilen geschrieben, so dass der erste auf der oberen, der zweite auf der unteren, der dritte Buchstabe wieder auf der oberen Zeile steht und so weiter. Der Geheimtext entsteht, indem abschließend die Zeichenkette der unteren Zeile an die der oberen Zeile angefügt wird. (aus Wikipedia)

Beispiel:

Klartext: `Gartenzaun`

Verfahren:

```
G r e z u
 a t n a n
```

Geheimtext: `Grezuatnan`

Aufgabe: Definieren und implementieren Sie eine Klasse `GartenzaunTransposition`, die das folgende Interface `Verschluesselung` mit dem Verfahren der Gartenzaum-Transposition implementiert.

```
interface Verschluesselung {

     // liefert den verschluesselten Text
     String verschluesseln(String klartext);

     // liefert den entschluesselten Text
     String entschluesseln(String geheimtext);
}
```

Aufgabe 20:

Teilaufgabe (a):

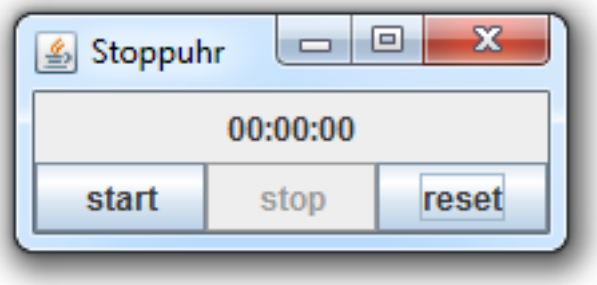
Implementieren Sie eine ADT-Klasse `Clock`, die Uhrzeiten von Stoppuhren (siehe auch Aufgabe 3) repräsentiert. Intern sollen dabei Stunden (0-23), Minuten (0 – 59) und Sekunden (0 – 59) verwaltet werden. Beachten Sie dabei die bekannten Umrechnungsregeln für Stunden, Minuten und Sekunden (zum Beispiel: 61 Minuten = 1 Stunde + 1 Minute). Die Klasse soll folgende Methoden zur Verfügung stellen:

- Einen Default-Konstruktor, der ein Clock-Objekt mit 00:00:00 initialisiert.
- Einen Konstruktor, der ein Clock-Objekt mit Werten für die Stunden, Minuten und Sekunden initialisiert, die ihm als Parameter übergeben werden.
- Einen Copy-Konstruktor.
- Eine Methode `clone` zum Klonieren eines Clock-Objektes (Überschreiben der von der Klasse `Object` geerbten Methode `clone`).
- Eine Methode `equals` zum Überprüfen der Gleichheit zweier Clock-Objekte (Überschreiben der von der Klasse `Object` geerbten Methode `equals`).
- Eine Methode `toString`, die eine String-Repräsentation des aktuellen Clock-Objektes liefert (Überschreiben der von der Klasse `Object` geerbten Methode `toString`). Die String-Repräsentation soll dabei folgendermaßen aufgebaut sein: `hh:mm:ss` (bspw. 23:08:34 für 23 Stunden, 8 Minuten und 34 Sekunden).
- Eine Methode `addSeconds`, die eine als Parameter übergeben Anzahl an Sekunden zur aktuell repräsentierten Uhrzeit addiert.
- Eine Klassenmethode `add`, die zwei Clock-Objekte als Parameter übergeben bekommt und ein Clock-Objekt erzeugt und liefert, das die Summe der Uhrzeiten der beiden Parameter-Objekte repräsentiert.
- Eine Methode `reset`, die die interne Uhrzeit auf 00:00:00 zurücksetzt.

Teilaufgabe (b):

Entwickeln Sie eine Stoppuhr als GUI-Applikation, die die ADT-Klasse `Clock` nutzt.

Die GUI besteht dabei aus einem Fenster, in dem (entsprechend obiger Abbildung) im oberen Teil die von einem Clock-Objekt repräsentierte Uhrzeit (anfangs 00:00:00) angezeigt wird. Im unteren Teil gibt es drei Buttons (start, stop, reset) zum Bedienen der Stoppuhr.



Klickt der Benutzer den start-Button, wird die Stoppuhr gestartet, d.h. nach Verstreichen jeder realen Sekunde wird die Uhr bzw. die Uhranzeige automatisch um eine Sekunde erhöht.

Klickt der Benutzer den stop-Button, wird die Stoppuhr gestoppt.

Klickt der Benutzer den reset-Button, wird die Ausgangssituation wieder hergestellt.

Beachten Sie: Wenn die Stoppuhr läuft, soll es nicht möglich sein, den start-Button zu klicken. Wenn die Stoppuhr nicht läuft, soll es nicht möglich sein, den stop-Button zu drücken (Methode `setEnabled`).

Hinweis: Nutzen Sie zum automatischen Erhöhen der Uhrzeit die Klassen `java.util.Timer` und `java.util.TimerTask`. Schauen Sie sich in der JDK-Dokumentation an, was diese Klassen leisten und wie sie zu nutzen sind.

Aufgabe 21:

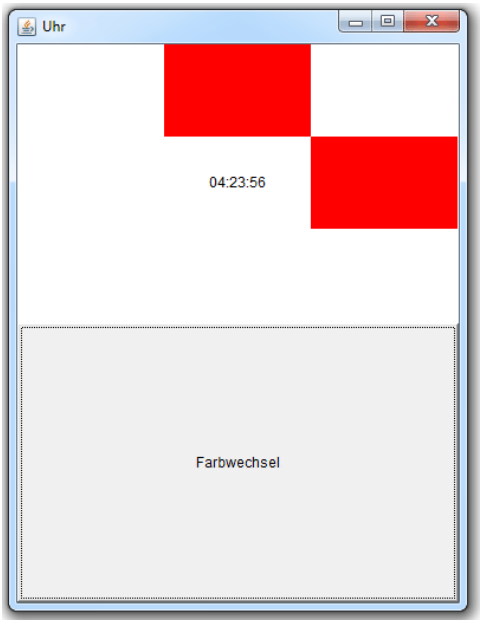
Teilaufgabe (a): Implementieren Sie eine neue GUI-Komponentenklasse namens `DigitalClock`, die von `java.awt.Panel` abgeleitet sein soll. Eine DigitalClock besteht dabei aus 3 * 3 Teilbereichen. Diese Teilbereiche enthalten folgende Inhalte:

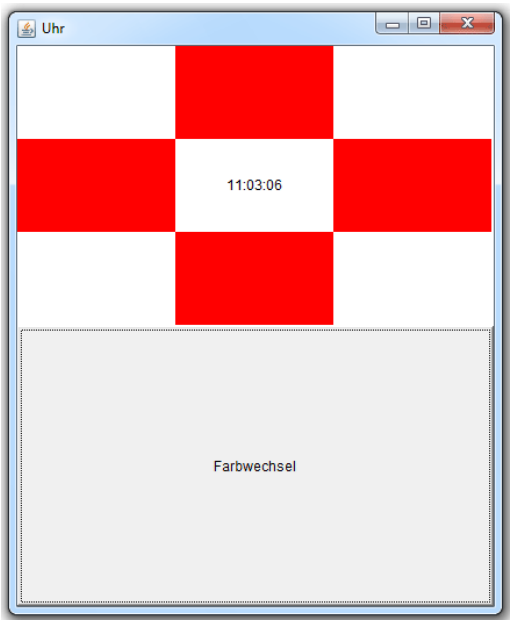
- In der Mitte befindet sich ein ClockLabel.
- Die Teilbereiche links oben, links unten, rechts oben und rechts unten haben keine Bedeutung (leere weiße Panel)
- Teilbereich oberhalb des ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 1 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.
- Teilbereich rechts vom ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 4 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.
- Teilbereich unterhalb des ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 7 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.
- Teilbereich links vom ClockLabel: ein Panel mit farbigem (Standardfarbe ist rot) Hintergrund zwischen 10 Uhr (einschließlich) und 0 Uhr (ausschließlich), ansonsten weiß.

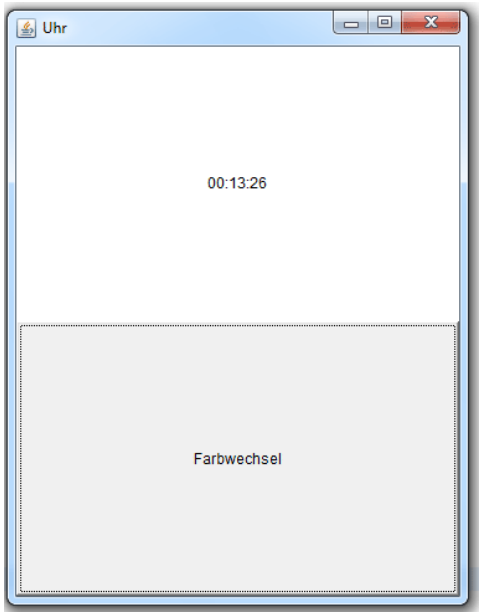
Nutzen Sie zur Realisierung der zeitbedingten Farbumstellungen innerhalb der Klasse `DigitalClock` die Möglichkeit, ClockListener bei dem ClockLabel zu registrieren (siehe API-Dokumentation).

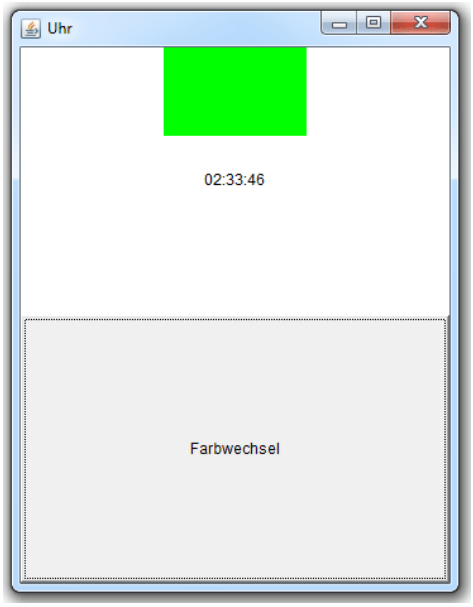
Teilaufgabe (b): Implementieren Sie dann eine Java-GUI-Anwendung mit einem Fenster, das, wie in den folgenden Abbildungen skizziert, eine DigitalClock und einen Button mit der Beschriftung „Farbwechsel“ enthalten soll. Bei Klick auf den Farbwechsel-Button soll die Farbe der farbig dargestellten Panel der DigitalClock wechselweise zwischen rot und grün umgeschaltet werden können.

Beispiele:



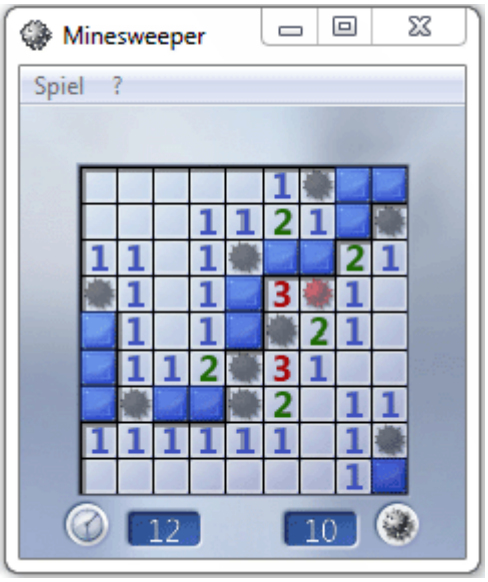






Aufgabe 22:

Sie kennen sicher das Spiel *Minesweeper*. Es ist ein Computerspiel, bei dem ein Spieler durch logisches Denken herausfinden muss, hinter welchen Feldern einer N * M-Matrix Mienen versteckt sind.

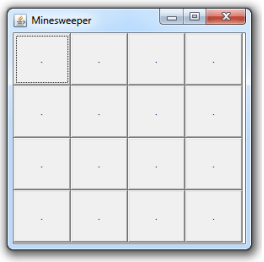


Während des Spiels wird beim Anklicken eines Feldes angezeigt, wie viele Mienen auf Nachbarfeldern platziert sind. Jedes Feld hat dabei 8 Nachbarfelder, Felder an den Rändern entsprechend weniger. Ziel des Spiels ist es, alle sicheren, also nicht mit Bomben belegten Felder zu finden. Sobald man auf ein Mienenfeld klickt, hat man verloren.

Implementieren Sie das Minesweeper-Spiel als Java-GUI-Anwendung. Ausgangspunkt ist eine im Quellcode vorgegebene (aber prinzipiell beliebige) M*N-Minesweeper-Lösungmatrix mit `int`-Werten, wobei eine Bombe durch den Wert `-1` repräsentiert wird, zum Beispiel:

```
final int[][] feld = { { -1, 1, 0, 0 },
                       { 2, 2, 1, 0 },
                       { 1, -1, 1, 0 },
                       { 1, 1, 1, 0 } };
```

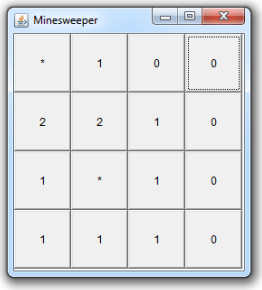
Die einzelnen Felder des Minesweeper-Spiels werden in der GUI durch Buttons repräsentiert. Anfangs enthalten alle Buttons das Label „.“.



Klickt der Benutzer auf ein Bombenfeld, ist das Spiel beendet und der Benutzer hat verloren. In diesem Fall soll das Label des Buttons durch das Label „*“ ersetzt werden. Außerdem sollen alle Buttons für Benutzerinteraktionen gesperrt werden (Button-Methode `setEnabled`).

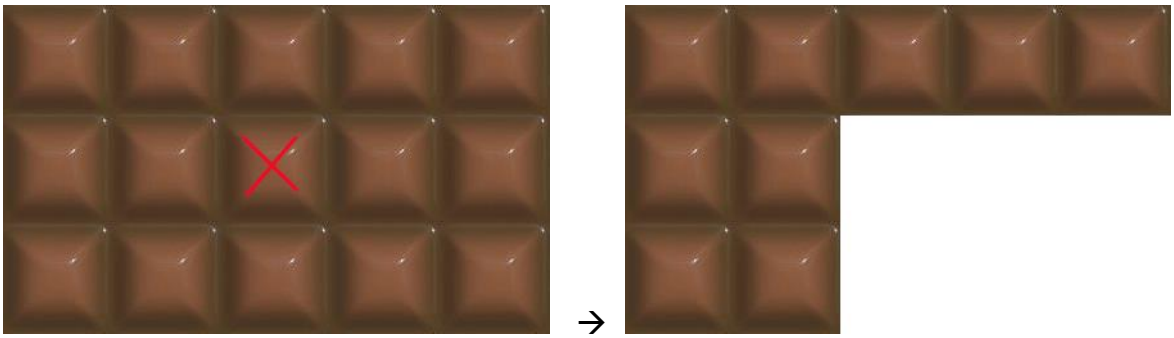
Klickt der Benutzer auf ein Nicht-Bombenfeld, soll das Label des Buttons durch den entsprechenden Wert des zugehörigen Matrix-Elementes ersetzt werden (also die Anzahl an Bomben auf Nachbarfeldern).

Beim Anklicken des letzten Nicht-Bombenfeldes, ist das Spiel beendet und der Benutzer hat gewonnen. In diesem Fall sollen unmittelbar alle Bombenfelder offengelegt werden, d.h. die Label der entsprechenden Buttons sollen durch das Label „`*`“ ersetzt werden.



Aufgabe 23:

Chomp ist ein Spiel für 2 Spieler. Gegeben eine Tafel Schokolade in Form eines rechteckiges Gebietes der Größe N * M, das aus einzelnen gleich großen Stückchen besteht. Die beiden Spieler essen abwechselnd ein (noch existierendes) Stückchen und damit gleichzeitig auch automatisch alle Stücken, die sich in Reihen darunter und Spalten rechts davon befinden. Wer das letzte Stückchen isst, bekommt Bauchschmerzen und verliert. Wer sich nicht an die Regeln hält, verliert ebenfalls unmittelbar.



Aufgabe:

Implementieren Sie das Chomp-Spiel, so dass es Menschen und Computer als Consolen-Spiel (ohne GUI) gegeneinander spielen können. Erweitern Sie dabei die folgende vorgegebene Klasse `Chomp`; genauer: Implementieren Sie die fehlende Methode `spielen` der Klasse `Chomp`.

Implementieren Sie eine `Spieler`-Klasse, die einen menschlichen Spieler repräsentiert. Eingaben von Spielzügen sollen über die Console erfolgen.

Implementieren Sie weiterhin eine `Spieler`-Klasse, die einen Computer-Spieler repräsentiert. Sie brauchen dabei keine Gewinnstrategie implementieren. Liefern Sie einfach zufällig ermittelte (aber korrekte!) Spielzüge.

Passen Sie die Zeilen 3 und 4 der `main`-Funktion der Klasse `Chomp` so an, dass ein Mensch gegen einen Computer-Spieler spielt.

Wichtig:

Sie dürfen keine Änderungen (sondern ausschließlich Erweiterungen) an den vorgegebenen Klassen vornehmen; mit Ausnahme der Implementierung der Methode `spielen` und der beiden Anweisungen in den Zeilen 3 und 4 der `main`-Funktion.

```
/**
 * Repraesentation von Spielzuegen
 */
class Spielzug {
	private int reihe; // Index der Reihe (ab 0)
	private int spalte; // Index der Spalte (ab 0)

	public Spielzug(int reihe, int spalte) {
		this.reihe = reihe;
		this.spalte = spalte;
	}

	public int getReihe() {
		return this.reihe;
	}

	public int getSpalte() {
		return this.spalte;
	}

	@Override
	public String toString() {
		return "Zug: r=" + this.reihe + "; s=" + this.spalte;
	}
}

/**
 * Schnittstelle fuer Chomp-Spieler; auf der Basis dieses Interfaces
 * sollen Sie einen menschlichen Spieler und einen Computer-Spieler
 * implementieren
 */
interface Spieler {
	/**
	 * liefert den naechsten Spielzug des Spielers
	 *
	 * @param schokolade
	 *            die aktuelle Schokolade (darf nicht veraendert werden!)
	 * @return der naechste Spielzug des Spielers
	 */
	public Spielzug naechsterSpielzug(boolean[][] schokolade);
}

/**
 * Chomp-Spiel
 */
class Chomp {

	// die beiden Spieler; Index 0 = Spieler 1; Index 1 = Spieler 2
	Spieler[] spieler;

	// die Schokolade; true bedeutet: Stueckchen existiert noch
	boolean[][] schokolade; // die Schokolade

	// initialisiert ein Chomp-Spiel
	public Chomp(int anzahlReihen, int anzahlSpalten, Spieler spieler1,
			Spieler spieler2) {
```

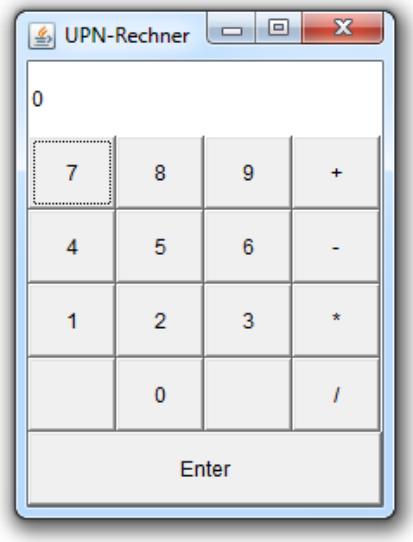
```
            this.spieler = new Spieler[2];
            this.spieler[0] = spieler1;
            this.spieler[1] = spieler2;
            this.schokolade = new boolean[anzahlReihen][anzahlSpalten];
            for (int r = 0; r < anzahlReihen; r++) {
                  for (int s = 0; s < anzahlSpalten; s++) {
                        this.schokolade[r][s] = true;
                  }
            }
      }

      // diese Methode muessen Sie implementieren
      public void spielen() {
            // - es beginnt Spieler 1
            // - Spielfeld ausgeben
            // - Endlosschleife:
            //    - Ausgabe des aktuellen Spielers
            //    - Spielzug beim aktuellen Spieler erfragen
            //          (ueber das interface Spieler!)
            //    - Spielzug ausgeben
            //    - Spielzug ueberpruefen (ggfl. Verlierer bekanntgeben
             //           und Ende)
            //    - Spielzug ausfuehren
             //    - Spielende ueberpruefen (ggfl. Verlierer bekanntgeben
             //           und Ende)
            //    - Spielfeld ausgeben
            //    - Spielerwechsel
      }
}

public class UE20Aufgabe23 {
      public static void main(String[] args) {
            final int anzahlReihen = 3;
            final int anzahlSpalten = 5;
            Spieler spieler1 = new Mensch();
            Spieler spieler2 = new Computer();
            Chomp chomp =
             new Chomp(anzahlReihen, anzahlSpalten, spieler1, spieler2);
            chomp.spielen();
      }
}
```

Aufgabe 24:

In dieser Aufgabe sollen Sie einen UPN-Rechner mit interaktiver GUI implementieren. Bei der UPN (Umgekehrte Polnische Notation) werden eingelesene Zahlen jeweils oben auf den Stapel gelegt. Wird ein Operator eingegeben, werden die beiden oberen Elemente vom Stapel entfernt, darauf die Operation angewendet, das Ergebnis ausgegeben und das Ergebnis auf den Stapel gelegt.

Teilaufgabe A:

Bilden Sie mit den Ihnen bekannten Java-AWT-Klassen exakt folgende GUI nach:



In der obersten Zeile wird die aktuelle Zahl angezeigt. Darunter befindet sich eine Menge an Buttons, die die einzelnen Ziffern und Operationen repräsentieren.

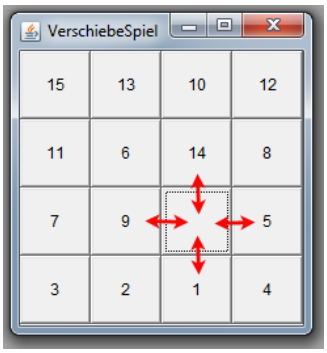
Teilaufgabe B:

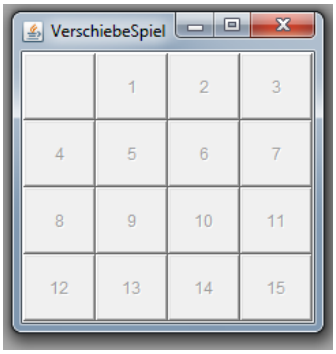
Implementieren Sie das „Verhalten“ des UPN-Rechners:

- Eingabe einer Ziffer → Beginn einer neuen oder Fortsetzung einer existierenden Zahl und Anzeige der aktuellen Zahl im oberen Label.
- Eingabe von „ENTER“ → die aktuelle Zahl wird auf den Stack gelegt.
- Eingabe eines Operators → Ausführung der Operation mit den beiden obersten Stack-Einträgen und Anzeige der berechneten Zahl im oberen Label.

Aufgabe 25:

Sie alle kennen aus Ihrer Kindheit das so genannte Slider- oder Verschiebe-Spiel. Es ist ein Spiel für eine Person. Das Spielgerät ist eine Tafel mit N * M Feldern. In dieser Tafel sind auf N * M - 1 Feldern Plättchen mit den Ziffern 1 bis N * M - 1 platziert. Ein Feld ist leer. Die Plättchen sind verschiebbar. Gegeben eine bestimmte Ausgangsstellung der Plättchen ist es das Ziel, die Plättchen in der Reihenfolge 1 bis 15 anzuordnen (das Feld oben links soll leer sein).





Ihre Aufgabe ist es, eine interaktive GUI-Anwendung zum Spielen des Slider-Spiels zu implementieren.

Hinweise und Teilaufgaben:

Die Ausgangsstellung soll über eine int-Matrix im Code festgelegt werden. In der Matrix stehen die Zahlen von 0 bis N*M-1. Die 0 repräsentiert das fehlende Plättchen. Für die obige Abbildung links entspricht das:

```
int[][] matrix = { { 15, 13, 10, 12 },
                   { 11, 6, 14, 8 },
                   { 7, 9, 0, 5 },
                    { 3, 2, 1, 4 } };
```

Sie dürfen nicht davon ausgehen, dass die Matrix immer aus 16 Elementen besteht. Sie kann prinzipiell aus N Reihen und M Spalten bestehen, mit N > 1 und M > 1.

Im Fenster werden die N*M Felder durch N*M Buttons repräsentiert. Die Buttons werden mit den entsprechenden Zahlen der Matrix beschriftet. Der Button, der das fehlende Plättchen repräsentiert, bleibt leer.

Beim Klick auf einen dem leeren Button angrenzenden Nachbar-Button (in der obigen Abbildung die Buttons 9, 14, 5 und 1) sollen der angeklickte und der leere Button „getauscht“ werden. Realisieren Sie einen Tausch durch einen Wechsel der Button-Beschriftung (Methoden `getLabel` und `setLabel`). Klicks auf andere Buttons bleiben ohne Wirkung.

Ist der Endzustand erreicht (siehe Abbildung rechts) sollen alle Buttons disabled werden, d.h. nicht mehr anklickbar sein (Methode `setEnabled`).

Aufgabe 26:

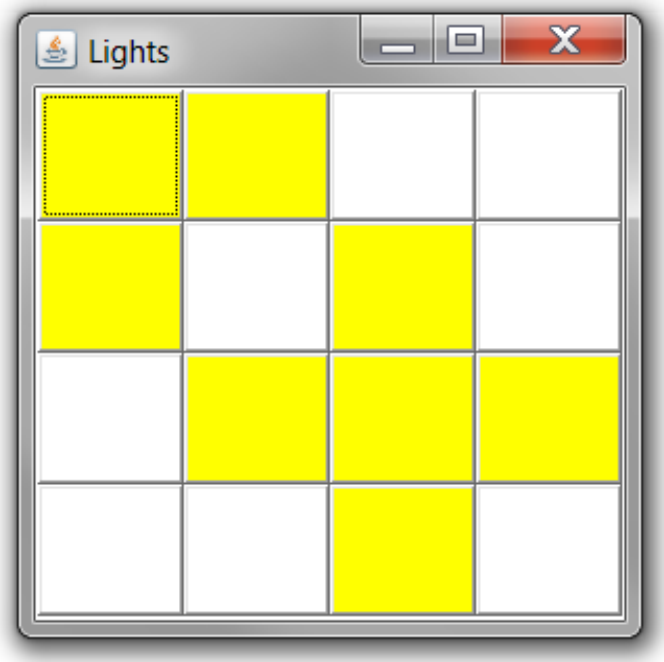
Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das ein menschlicher Spieler am Computer das Knobelspiel *Lampen* spielen kann.

Regeln:

Das Spiel besteht aus einem quadratischen Spielfeld der Größe n ($2 < n < 10$), das aus einzelnen Zellen besteht, die Lampen repräsentieren. Lampen können sich im Zustand *an* oder *aus* befinden. Anfangs sind alle Lampen im Zustand *aus*. In jedem Spielzug wählt der Spieler eine Zelle aus, deren Zustand umgeschaltet werden soll (von *aus* in *an* oder umgekehrt). Gleichzeitig werden aber auch die Nachbarlampen oberhalb, unterhalb, links und rechts von der entsprechenden Lampe umgeschaltet (insofern sie existieren). Ziel (und Ende) des Knobelspiels ist es, den Zustand zu erreichen, dass alle Lampen angeschaltet sind.

Aufgabe:

Ihre Aufgabe ist es, eine interaktive GUI für das Lampen-Spiel zu entwickeln. Realisieren Sie die einzelnen Lampen jeweils durch einen Button. Repräsentieren Sie die beiden Lampenzustände durch verschiedene Background-Farben der Buttons.



Aufgabe 27:

In dieser Aufgabe geht es um die Modellierung von unendlichen Zahlenfolgen. Als *Zahlenfolge* wird in der Mathematik eine Auflistung von endlich oder unendlich vielen fortlaufend nummerierten Objekten bezeichnet. Die Objekte sind in dieser Aufgabe `int`-Werte.

Teilaufgabe 1:

Definieren Sie ein Interface **`Sequence`** für allgemeine unendliche Zahlenfolgen mit int-Werten. `Sequence` hat eine Methode:

- `getNext`: liefert die nächste Zahl der Folge.

Teilaufgabe 2:

Die natürlichen Zahlen (1, 2, 3, …) sind eine konkrete unendliche Zahlenfolge. Definieren Sie eine Klasse **`Naturals`**, die das Interface `Sequence` implementiert. Der erste Aufruf von `getNext` für ein `Naturals`-Objekt liefert 1, der nächste 2, dann 3 und so weiter.

Teilaufgabe 3:

Ein „Filter“ ist eine Zahlenfolge, die keine eigene Zahlenquelle enthält. Ein Filter „ernährt“ sich stattdessen von einer anderen Zahlenfolge, deren Elemente er entweder durchlässt oder absorbiert.

Definieren Sie eine abstrakte Klasse **`Filter`** (`abstract class Filter implements Sequence`) mit einem Konstruktor, dem als Parameter ein

`Sequence`-Objekt – die Zahlenquelle – übergeben bekommt. Die Zahlenquelle wird in einem Attribut gespeichert.

Teilaufgabe 4:

Leiten Sie von der abstrakten Klasse `Filter` eine konkrete Filter-Klasse **`ZapMultiples`** ab. Deren Konstruktor erwartet neben der Zahlenquelle als weiteren Parameter eine Basiszahl. Der Filter absorbiert die Basiszahl und alle ganzzahligen Vielfachen der Basiszahl und gibt den Rest weiter.

Beispiel 1: Der mehrfache Aufruf der Methode `getNext` von `new ZapMultiples(3, new Naturals())` würde folgende Zahlen liefern: 1, 2, 4, 5, 7, 8, 10, …

Beispiel 2: Der mehrfache Aufruf der Methode `getNext` von `new ZapMultiples(2, new ZapMultiples(3, new Naturals()))` würde folgende Zahlen liefern: 1, 5, 7, 11, 13, … (Absorption aller Vielfachen von 2 und 3)

Teilaufgabe 5:

Leiten Sie von der abstrakten Klasse `Filter` eine weitere konkrete Filter-Klasse **`Primes`** ab, die Primzahlen in aufsteigender Reihenfolge berechnet. Der mehrfache Aufruf der Methode `getNext` von `new Primes()` sollte also folgende Zahlen liefern: 2, 3, 5, 7, 11, 13, 17, …

Nutzen Sie dabei zur Ermittlung der Primzahlen den folgenden Algorithmus („Sieb des Eratosthenes“):

1. Nimm die Folge der natürlichen Zahlen und entferne die 1.
2. Die erste Zahl *p* des Restes der Folge ist eine Primzahl.
3. Entferne *p* und alle Vielfachen von *p* aus der Folge.
4. Zurück zu Schritt 2.

Implementieren Sie diesen Algorithmus mit den in den Teilaufgaben 2 und 4 implementierten Klassen!

Quellenangabe: Die Aufgabe stammt aus dem Buch „Programmieren mit Java“ von R. Schiedermeier, erschienen im Verlag Pearson Studium.

Aufgabe 28:

Das folgende Java-Programm gibt die einzelnen Buchstaben einer Zeichenfolge in sortierter Reihenfolge aus.

```
import java.util.ArrayList;
import java.util.Collections;

public class Buchstaben {

     public static void main(String[] args) {
           String str =
```

```
"rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz";
          ArrayList<Buchstabe> buchstaben =
         berechneBuchstabenHaeufigkeit(str);
          Collections.sort(buchstaben);
          for (Buchstabe b : buchstaben) {
                System.out.print(b);
          }
    }
}

class Buchstabe implements Comparable<Buchstabe> {
    private char buchstabe;
    private int haeufigkeit;

    public Buchstabe(char buchstabe) {
          this.buchstabe = buchstabe;
          this.haeufigkeit = 1;
    }

    @Override
    public boolean equals(Object obj) {
          return this.buchstabe == ((Buchstabe) obj).buchstabe;
    }

    @Override
    public String toString() {
          String res = "";
          for (int i = 0; i < this.haeufigkeit; i++) {
                res += this.buchstabe;
          }
          return res;
    }

    public void inkr() {
          this.haeufigkeit++;
    }

    public char getBuchstabe() {
          return this.buchstabe;
    }

    public int getHaeufigkeit() {
          return this.haeufigkeit;
    }

    @Override
    public int compareTo(Buchstabe b) {
          return this.buchstabe - b.buchstabe;
    }
}
```

Für den eingetragenen Beispiel-String („rindfleisch…") erfolgt bspw. die Ausgabe:

```
aaaabbbccdeeeeeeeeeeffgggggghhiiiiklnnnnnrrrrrsssssttttttuuuuwzüü
```

Was in dem Programm noch fehlt, ist die Implementierung der Methode `berechneBuchstabenHaeufigkeit`. Die Implementierung dieser Methode ist Ihre Aufgabe. Die Methode ermittelt und liefert eine `ArrayList` mit so vielen Elementen vom Typ `Buchstabe`, wie es unterschiedliche Buchstaben im als Parameter

übergebenen String gibt. Jedes `Buchstabe`-Element speichert dabei den `char`-Wert des Buchstabens sowie die Häufigkeit des Buchstabens im String.

Aufgabe 29:

Gegeben sei folgendes Interface

```
interface IntTester {
     boolean test(int value);
}
```

Man sagt, der test-Methode als Parameter übergebene int-Werte bestehen den Test genau dann, wenn die test-Methode *true* liefert.

Teilaufgabe 1:

Definieren und implementieren Sie eine Klasse `OddValueTester`, die das Interface `IntTester` implementiert. int-Werte sollen den Test der Klasse `OddValueTester` bestehen, wenn sie ungerade sind.

Teilaufgabe 2:

Leiten Sie eine Klasse `IntArrayList` von der Klasse `java.util.ArrayList` ab. Objekte der Klasse `IntArrayList` sollen int-Werte speichern können.

Definieren Sie in der Klasse `IntArrayList` eine Instanzmethode `sum`, die die Summe aller im aufgerufenen Objekt gespeicherten int-Werte berechnet und als Ergebnis liefert.

Definieren Sie in der Klasse `IntArrayList` weiterhin eine Instanzmethode namens `filter`. Diese habe einen Parameter vom Typ `IntTester` und ihr Funktionstyp ist `IntArrayList`. Die Methode soll ein `IntArrayList`-Objekt erzeugen und zu diesem alle die in dem aufgerufenen `IntArrayList`-Objekt gespeicherten int-Werte, die den Test des als Parameter übergebenen `IntTester`-Objektes bestehen, hinzufügen und dieses dann als Ergebnis liefern.

Beispiel:

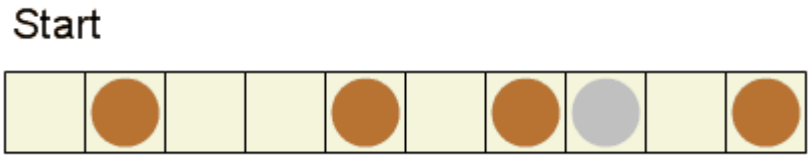
Das folgende kleine Testprogramm sollte „25“ auf den Bildschirm ausgeben:

```
IntArrayList list = new IntArrayList();
for (int i = 1; i <= 10; i++) {
     list.add(i);
}
System.out.println(list.filter(new OddValueTester()).sum());
```

Aufgabe 30:

Implementiert werden soll in dieser Aufgabe das sogenannte Silver-Dollar-Game.

Das "Silver-Dollar-Game" ist ein Spiel für zwei Spieler. Es wird auf einem Spielbrett gespielt, das aus einer Reihe an n Feldern besteht. Auf jedem der Felder liegt anfangs entweder kein oder ein Dollar. Genau ein Dollar ist dabei ein silberner Dollar. Die anderen sind golden.



Die beiden Spieler absolvieren immer abwechselnd Spielzüge. Es besteht Zugzwang. Dabei gibt es zwei Typen von Spielzügen:

- R-Spielzug: Der Spieler wählt ein Feld aus, auf dem sich ein Dollar befindet. Er verschiebt dann den Dollar um ein oder mehrere Felder nach links. Dabei darf der Dollar jedoch nicht über andere Dollars hinweg verschoben oder außerhalb des Spielbrettes platziert werden.
- S-Spielzug: Der Spieler entfernt den am weitesten links platzierten Dollar vom Spielbrett.

Das Spiel ist beendet, wenn ein Spieler einen fehlerhaften Zug durchführt oder wenn der silberne Dollar vom Spielbrett entfernt wurde. Gewinner ist im ersten Fall der Gegenspieler und im zweiten Fall der Spieler, der den silbernen Dollar entfernt hat.

Aufgabe: Implementieren Sie ein Java-Programm mit einer GUI, mit dem zwei menschliche Spieler gegeneinander das Silver-Dollar-Game spielen können.

Das anfängliche Spielbrett kann dabei im Sourcecode in der folgenden Form fest vorgegeben werden. Ihr Programm muss prinzipiell aber auch mit einer anderen Anzahl an Feldern und einer anderen anfänglichen Belegung mit Dollars funktionieren:

```
char[] brett = { '.','1','.','.','1','.','1','2','.','1'};
```

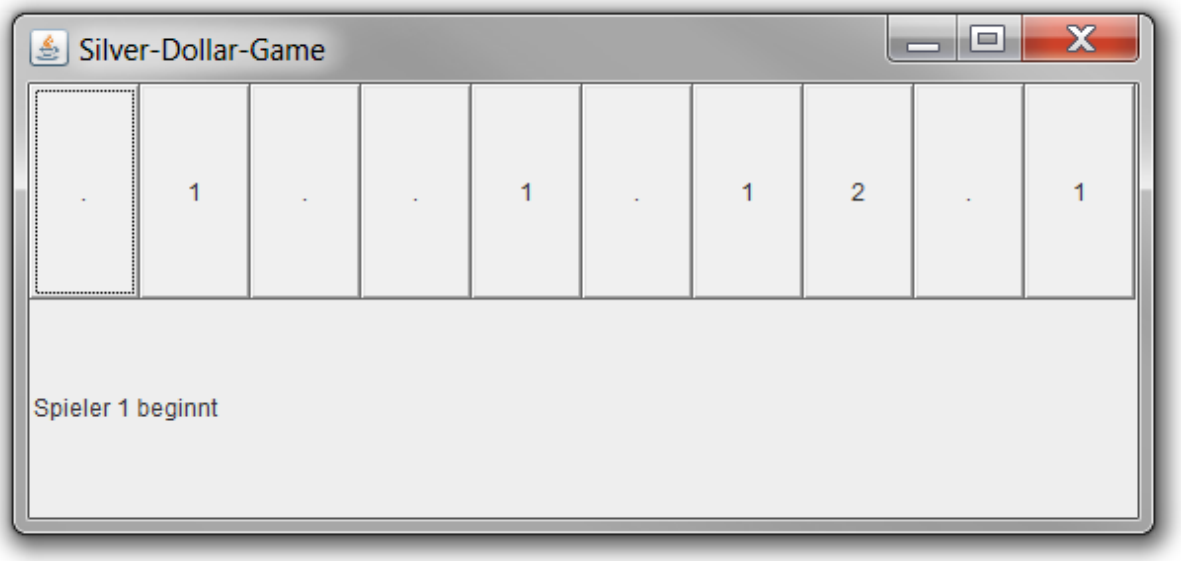
Dabei bedeuten:

`'.'`: leeres Feld

`'1'`: goldener Dollar

`'2'`: silberner Dollar

Die GUI des Spiels soll folgende Gestalt haben:



Im oberen Bereich des Fensters werden die einzelnen Felder des Spiels durch Buttons repräsentiert. Unten im Fenster befindet sich ein Label für Textausgaben.

Gespielt wird gemäß der obigen Regeln:

- Klickt der Benutzer auf ein leeres Feld, so hat er unmittelbar verloren (ungültiger Spielzug)
- Klickt der Benutzer auf das Dollar-Feld, das am weitesten links ist, so wird der Dollar entfernt (S-Spielzug)
- Klickt der Benutzer auf ein anderes Dollar-Feld (R-Spielzug), muss er anschließend entsprechend der Regeln einen weiteren Button anklicken, auf den der Dollar umplatziert werden soll. Hält er sich an die Regeln, wird der Dollar entsprechend umplatziert. Klickt der Benutzer auf ein unerlaubtes Feld, so hat er unmittelbar verloren.
- Bei Spielende (Sieg oder ungültiger Spielzug) sollen alle Buttons disabled (d.h. nicht mehr anklickbar) werden.

Im Label unten im Fenster werden Nachrichten angezeigt. Wählen Sie selbst aussagekräftige Nachrichten.

Aufgabe 31:

Gegeben sei folgende Klasse *Element* zur Repräsentation chemischer Elemente:

```
public class Element {

    // Symbol des Elementes
    protected String symbol;

    // Ordnungszahl = Anzahl der Protonen im Atomkern eines chemischen
    // Elements
    protected int ordnungszahl;
```

```
    public Element(String kuerzel, int ordnungszahl) {
          this.symbol = kuerzel;
          this.ordnungszahl = ordnungszahl;
    }

    public String toString() {
          return symbol + "(" + ordnungszahl + ")";
    }
}
```

Sie bekommen nun in einer Anwendung ein Array mit *Element*-Objekten übergeben und müssen die Elemente des Arrays nach der Größe der Ordnungszahl sortieren. Die Klasse *java.util.Arrays* bietet eine Klassenmethode *sort* zum Sortieren eines Arrays, das als Parameter übergeben wird (`Object[]`). Problem ist, dass die Methode *sort* eine Exception wirft, wenn die Elemente des Arrays nicht das Interface *java.lang.Comparable* implementieren:

```
public interface Comparable<T> {
  // vergl. das this-Objekt mit dem als Parameter übergebenen Objekt o;
  // liefert einen negativen Wert, wenn das this-Objekt kleiner ist als o;
  // liefert einen positiven Wert, wenn das this-Objekt größer ist als o;
  // liefert 0, wenn das this-Objekt gleich groß ist wie o
  int compareTo(T o);
}
```

Aufgabe: Leiten Sie eine Klasse *ChemieElement* von der Klasse *Element* ab, die das Interface *Comparable* implementiert. Dabei soll gelten: Ein chemisches Element e1 ist kleiner bzw. größer als ein chemisches Element e2, wenn die Ordnungszahl von e1 kleiner bzw. größer als die Ordnungszahl von e2 ist. Das folgende Programm

```
public class Chemie {

    public static void main(String[] args) {
          Element[] elemente = getBeispielElemente();
          java.util.Arrays.sort(elemente);
          for (Element e : elemente) {
                System.out.println(e);
          }
    }

    static Element[] getBeispielElemente() {
          Element[] elemente = new Element[5];
          elemente[0] = new ChemieElement("Ca", 20);
          elemente[1] = new ChemieElement("Mg", 12);
          elemente[2] = new ChemieElement("H", 1);
          elemente[3] = new ChemieElement("O", 8);
          elemente[4] = new ChemieElement("Pd", 46);
          return elemente;
    }
}
```

sollte dann die folgende Ausgabe produzieren:

```
H(1)
O(8)
Mg(12)
```

```
Ca(20)
Pd(46)
```

Aufgabe 32:

Wir haben in der Vorlesung eine ADT-Klasse *Stack* als Implementierung des abstrakten Datentyps Stack kennen gelernt. Auf einen Stack können nur oben Elemente drauf gelegt werden (Methode *push*) und immer nur das oberste Element eines Stacks kann zugegriffen und entfernt werden (Methode *pop*).

Ein *DoubleSideStack* ist ein Stack mit einer linken und einer rechten Seite. Auf einen DoubleSideStack können nur links und rechts Elemente hinzugefügt werden (*pushLeft*, *pushRight*) und es kann immer nur auf das am weitesten links und rechts liegende Element zugegriffen und entfernt werden (*popLeft*, *popRight*).

Implementieren Sie eine ADT-Klasse *DoubleSideStack*, die das folgende Interface *DoubleSideStackI* implementiert. Die Klasse soll weiterhin einen Default-Konstruktor und einen Copy-Konstruktor besitzen. Nutzen Sie als interne Datenstruktur eine `java.util.ArrayList<Integer>`.

```
public interface DoubleSideStackI {

  // liefert genau dann true, wenn sich kein Element auf dem DoubleSideStack
  // befindet
  public boolean isEmpty();

  // fuegt den uebergebenen Wert auf der rechten Seite des DoubleSideStacks an
  public void pushRight(int value);

  // entfernt das am weitesten rechts liegende Element aus dem DoubleSideStack
  // und liefert es als Methodenwert; ist der DoubleSideStack leer, wird eine
  // java.lang.IndexOutOfBoundsException als Unchecked-Exception geworfen
  public int popRight();

  // fuegt den uebergebenen Wert auf der linken Seite des DoubleSideStacks an
  public void pushLeft(int value);

  // entfernt das am weitesten links liegende Element aus dem DoubleSideStack
  // und liefert es als Methodenwert; ist der DoubleSideStack leer, wird eine
  // java.lang.IndexOutOfBoundsException als Unchecked-Exception geworfen
  public int popLeft();

  // liefert einen String, in dem die Werte der Elemente des DoubleSideStack
  // von links nach rechts enthalten sind, nach jedem Wert steht ein
  // Leerzeichen; Beispiel: "12 2 -34 "
  public String toString();

  // liefert einen neuen DoubleSideStack, der dieselben Werte in derselben
  // Reihenfolge wie der aufgerufene DoubleSideStack speichert; Achtung: Die
  // beiden DoubleSideStacks sollen unabhaengig voneinander sein, d.h. sie
  // sollen sich keine Elemente teilen!
  public DoubleSideStackI clone();

  // liefert genau dann true; wenn beide DoubleSideStacks dieselben Werte in
  // derselben Reihenfolge speichern
  public boolean equals(Object obj);
}
```

Das folgende Beispielprogramm

```
DoubleSideStack stack = new DoubleSideStack();
stack.pushLeft(23);
stack.pushLeft(-34);
stack.pushRight(5);
stack.pushLeft(-8);
stack.pushRight(67);
System.out.println(stack.toString());
stack.popLeft();
stack.popRight();
System.out.println(stack.toString());
```

Sollte folgende Ausgabe erzeugen:

```
-8 -34 23 5 67
-34 23 5
```

Aufgabe 33:

Münzen ist ein Spiel für N (> 2) Spieler mit den Nummern 0 bis N-1. Vor den Spielern liegt eine Reihe an Münzen. Die Münzen haben unterschiedliche Werte. Die Werte sind sichtbar. Ziel von jedem Spieler ist es, eine höhere Anzahl an Münzwerten zu sammeln als die anderen Spieler. Genommen wird der Reihe nach. Es besteht Zugzwang. Ein Spielzug sieht so aus, dass der Spieler, der an der Reihe ist, entweder die Münze ganz rechts oder die Münze ganz links nimmt. Das Spiel ist beendet, wenn alle Münzen genommen wurden. Sieger sind die Spieler mit der höchsten Anzahl an gesammelten Münzwerten.

Gegeben ist das folgende Programmfragment zum Spielen des Spiels Münzen an einer Konsole. Das Programm nutzt dabei einen *DoubleSideStack* aus Aufgabe 32:

```
public class Muenzen {

  // Konstanten sollen prinzipiell geaendert werden koennen, ohne den Code
  // aendern zu muessen!
  final static int ANZAHL_MUENZEN = 12;
  final static int ANZAHL_SPIELER = 3;
  final static int MIN_MUENZWERT = 1;
  final static int MAX_MUENZWERT = 6;

  public static void main(String[] args) {
    // speichert die Reihe von Muenzen
    DoubleSideStackI muenzen = initMuenzen();
    // speichert die Punkte pro Spieler
    int[] spieler = initSpieler();
    // aktueller Spieler (Index zum spieler-Array)
    int aktuellerSpielerIndex = 0;
    printMuenzenUndStand(muenzen, spieler);
    while (true) {
      printAktuellerSpieler(aktuellerSpielerIndex);
      spielzug(muenzen, spieler, aktuellerSpielerIndex);
      printMuenzenUndStand(muenzen, spieler);
      if (spielEnde(muenzen)) {
        siegerBekanntgeben(spieler);
        break;
      }
      aktuellerSpielerIndex = naechsterSpieler(aktuellerSpielerIndex);
    }
```

```
  }

  // initialisiert die Spieler; anfangs haben alle 0 Punkte
  static int[] initSpieler() { return new int[ANZAHL_SPIELER]; }

  // initialisiert die Muenzen (per Zufall)
  static DoubleSideStackI initMuenzen() {
    DoubleSideStackI muenzen = new DoubleSideStack();
    for (int i = 0; i < ANZAHL_MUENZEN; i++) {
      muenzen.pushRight(getRandomNumber(MIN_MUENZWERT, MAX_MUENZWERT));
    }
    return muenzen;
  }

  // liefert Zufallszahl zwischen min und max
  static int getRandomNumber(int min, int max) {
    return (int) (min + ((Math.random() * (max - min)) + 1));
  }


  // - gibt die Muenzen aus
  // - gibt die Punktestaende der einzelnen Spieler aus
  static   void   printMuenzenUndStand(DoubleSideStackI   muenzen,   int[]
spieler) {
    System.out.println("Muenzen: " + muenzen.toString());
    for (int i = 0; i < spieler.length; i++) {
      System.out.println("Spieler " + i + ": " + spieler[i]);
    }
  }

  // gibt aus, welcher Spieler als naechstes an der Reihe ist
  static void printAktuellerSpieler(int index) {
    System.out.println("Spieler " + index + " ist am Zug!");
  }

  // ueberprueft das Spielende
  static boolean spielEnde(DoubleSideStackI muenzen) {
    return muenzen.isEmpty();
  }

}
```

Aufgabe: Implementieren Sie die fehlenden Methoden:

(1) **naechsterSpieler**: liefert den Spieler-Index des Spielers, der als nächstes an der Reihe ist

(2) **siegerBekanntgeben**: ermittelt die Gewinner und gibt jeden Gewinner jeweils in der Form „Spieler <nummer> ist Gewinner“ auf die Konsole aus.

(3) **spielzug**: Durchführung eines Spielzugs:

- Zunächst wird der aktuelle Spieler gefragt, von welcher Seite (rechts oder links) er die Münze nehmen will.
- Die Münze wird von der entsprechende Seite entfernt
- Der Wert der entfernten Münze wird zum Punktestand des aktuellen Spielers addiert

Beispiel für Spielablauf (Benutzereingaben in <>):

```
Muenzen: 4 4 3 5 6 2 4 4 4 5 5 2
Spieler 0: 0
Spieler 1: 0
Spieler 2: 0
```

```
Spieler 0 ist am Zug!
Muenze von welcher Seite nehmen (r oder l)? <l>
Muenzen: 4 3 5 6 2 4 4 4 5 5 2
Spieler 0: 4
Spieler 1: 0
Spieler 2: 0
Spieler 1 ist am Zug!
...
Muenze von welcher Seite nehmen (r oder l)? <l>
Muenzen:
Spieler 0: 18
Spieler 1: 14
Spieler 2: 16
Spieler 0 ist Gewinner
```

Aufgabe 34:

Schauen Sie sich die folgende Klasse bzw. das folgende Interface *Observable* und *Observer* an:

```
public interface Observer {
  void update(Object info);
}

public class Observable {

  private java.util.ArrayList<Observer> observers =
    new java.util.ArrayList<Observer>();

  public void addObserver(Observer obs) {
    observers.add(obs);
  }

  public void notifyObservers(Object info) {
    for (Observer obs : observers) {
      obs.update(info);
    }
  }
}
```

Dahinter verbirgt sich das Konzept, dass ein Observable ein Objekt ist, das sich durch interessierte Observer-Objekte beobachten lässt. Wenn sich der Zustand des Observables ändert, informiert es darüber die interessierten Observer-Objekte.

Gegeben sei folgende Klasse *Thermometer*:

```
public class Thermometer extends Observable {

  public void messen() {
    while (true) {
      int grad = (int) (Math.random() * 40);
      this.notifyObservers(grad);
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
      }
    }
  }
}
```

In der Methode *messen* der Klasse *Thermometer* wird endlos jede Sekunde eine Zufallstemperatur ermittelt und den angemeldeten Observern mitgeteilt.

Teilaufgabe (a):

Implementieren Sie eine Klasse *Schule*, die das Interface *Observer* implementiert. Dem Konstruktor der Klasse *Schule* wird ein Thermometer-Objekt übergeben, bei dem sich das Schule-Objekt als Observer registriert (*addObserver*). Jedes Mal wenn das Schule-Objekt über eine Temperaturänderung informiert wird, soll es die neue Temperatur prüfen und bei mehr als 28 Grad den folgenden Text auf die Konsole ausgeben: „x Grad! Schule gibt Hitzefrei!“ (x = aktuelle Gradzahl).

Teilaufgabe (b):

Implementieren Sie eine Klasse *Freibad*, die das Interface *Observer* implementiert. Dem Konstruktor der Klasse *Freibad* wird ein Thermometer-Objekt übergeben, bei dem sich das Freibad-Objekt als Observer registriert (addObserver). Anfangs befindet sich das Freibad im Zustand geschlossen. Jedes Mal wenn das Freibad-Objekt über eine Temperaturänderung informiert wird, soll es die neue Temperatur prüfen. Ist es geschlossen und die Temperatur ist über 20 Grad soll es geöffnet werden und „x Grad! Warm genug! Freibad wird geoeffnet!“ auf die Konsole ausgegeben werden. Ist das Freibad geöffnet und die Temperatur ist unter 10 Grad soll es geschlossen und die „x Grad! Zu kalt! Freibad wird geschlossen!“ auf die Konsole ausgegeben werden (x = aktuelle Gradzahl).

Die Wetter-Simulation kann mit folgendem Programm gestartet werden:

```
public class Wetter {

  public static void main(String[] args) {
    Thermometer thermometer = new Thermometer();
    Schule schule = new Schule(thermometer);
    Freibad bad = new Freibad(thermometer);
    thermometer.messen();
  }
}
```

Eine mögliche Ausgabe wäre dann:

```
22 Grad! Warm genug! Freibad wird geoeffnet!
7 Grad! Zu kalt! Freibad wird geschlossen!
36 Grad! Schule gibt Hitzefrei!
36 Grad! Warm genug! Freibad wird geoeffnet!
4 Grad! Zu kalt! Freibad wird geschlossen!
25 Grad! Warm genug! Freibad wird geoeffnet!
32 Grad! Schule gibt Hitzefrei!
8 Grad! Zu kalt! Freibad wird geschlossen!
25 Grad! Warm genug! Freibad wird geoeffnet!
3 Grad! Zu kalt! Freibad wird geschlossen!
```

Aufgabe 35:

Wir haben in der Vorlesung eine ADT-Klasse *Stack* als Implementierung des abstrakten Datentyps Stack kennen gelernt. Auf einen Stack können nur oben Elemente drauf gelegt werden (Methode *push*) und immer nur das oberste Element eines Stacks kann zugegriffen und entfernt werden (Methode *pop*).

Ein *LRStack* ist ein Stack mit einer linken und einer rechten Seite. Auf einen LRStack können nur links und rechts Elemente hinzugefügt werden (*pushLeft*, *pushRight*) und es kann immer nur auf das am weitesten links und rechts liegende Element zugegriffen und entfernt werden (*popLeft*, *popRight*).

Implementieren Sie eine ADT-Klasse *LRStack*, die das folgende Interface *LRStackI* implementiert. Die Klasse soll weiterhin einen Default-Konstruktor und einen Copy-Konstruktor besitzen. Nutzen Sie als interne Datenstruktur eine `java.util.ArrayList<Integer>`.

```
public class EmptyException extends Exception {}

public interface LRStackI {
  // liefert genau dann true, wenn sich kein Element auf dem LRStack
  // befindet
  public boolean isEmpty();

  // fuegt den uebergebenen Wert auf der rechten Seite des LRStacks an
  public void pushRight(int value);

  // entfernt das am weitesten rechts liegende Element aus dem LRStack
  // und liefert es als Methodenwert; ist der LRStack leer, wird 0
  // geliefert
  public int popRight();

  // fuegt den uebergebenen Wert auf der linken Seite des LRStacks an
  public void pushLeft(int value);

  // entfernt das am weitesten links liegende Element aus dem LRStack
  // und liefert es als Methodenwert; ist der LRStack leer, wird 0
  // geliefert
  public int popLeft();

  // liefert einen String, in dem die Werte der Elemente des LRStack
  // von links nach rechts enthalten sind, nach jedem Wert steht ein
  // Leerzeichen; Beispiel: "12 2 -34 "
  public String toString();

  // liefert einen neuen LRStack, der dieselben Werte in derselben
  // Reihenfolge wie der aufgerufene LRStack speichert; Achtung: Die
  // beiden LRStacks sollen unabhaengig voneinander sein, d.h. sie
  // sollen sich keine Elemente teilen!
  public LRStackI clone();

  // liefert genau dann true; wenn beide LRStacks dieselben Werte in
  // derselben Reihenfolge speichern
  public boolean equals(Object obj);
}
```

Das folgende Beispielprogramm

```
LRStack stack = new LRStack();
stack.pushLeft(23);
stack.pushLeft(-34);
stack.pushRight(5);
stack.pushLeft(-8);
stack.pushRight(67);
System.out.println(stack.toString());
stack.popLeft();
stack.popRight();
System.out.println(stack.toString());
```

Sollte folgende Ausgabe erzeugen:

```
-8 -34 23 5 67
-34 23 5
```

Aufgabe 36:

Implementiert werden soll in dieser Aufgabe das Spiel *Sowing*. Sowing ist ein Brettspiel für 2 Spieler: Spieler 1 und Spieler 2. Gespielt wird auf einem Brett mit einer Reihe von N Feldern (hier N = 12). Auf jedem Feld liegen anfangs M Samen (hier M = 3). Spieler 1 spielt von links nach rechts, Spieler 2 spielt von rechts nach links. Beide Spieler ziehen abwechselnd. Es besteht Zugzwang. In jedem Zug muss ein Spieler den gesamten Inhalt eines nicht leeren Feldes in seine Zugrichtung auf die darauf folgenden Felder verteilen. Dabei wird auf jedes Feld ein Samen gelegt, bis alle Samen verteilt sind. Der Inhalt eines Feldes darf nur dann verteilt werden, wenn genug Felder in Zugrichtung liegen und der letzte Samen auf ein nicht leeres Feld fallen würde. Der Spieler, der als erster keinen Zug mehr machen kann, verliert. Ein Unentschieden ist nicht möglich.

Aufgabe: Implementieren Sie ein Java-Programm mit einer GUI, mit dem 2 menschliche Spieler gegeneinander das Spiel Sowing spielen können.

Die Parameter des Spiels können dabei im Sourcecode als Konstanten fest vorgegeben werden. Ihr Programm muss prinzipiell aber auch mit einer anderen Konfiguration funktionieren:

```
final static int ANZAHL_MULDEN = 12; // > 1
final static int SAMEN_PRO_MULDE = 3; // > 0
```

Die GUI des Spiels soll exakt folgende Gestalt haben:

Oben im Fenster befindet sich ein Label, das die Spielrichtung von Spieler 1 angibt. Darunter befinden sich in einer Reihe die Felder, die durch je einen Button repräsentiert werden, dessen Label die Anzahl der Samen auf dem jeweiligen Feld

angibt. Darunter wiederum befindet sich ein Label, das die Spielrichtung von Spieler 2 angibt. Ganz unten befindet sich ein Label für Nachrichten.

Gespielt wird gemäß der obigen Regeln:

- Im Nachrichten-Label wird jeweils bekannt gegeben, welcher Spieler am Zug ist.
- Ein Spielzug besteht aus dem Anklicken des Buttons, dessen Samen in der jeweiligen Spielrichtung verteilt werden sollen.
- Es sind immer alle Buttons anklickbar. Tätigt der aktuelle Spieler einen ungültigen Spielzug, wird er durch eine Nachricht im Nachrichten-Label darüber informiert und zu einem gültigen Spielzug aufgefordert.
- Bei Spielende wird im Nachrichten-Label der Verlierer bekannt gegeben, und es werden alle Buttons gesperrt (disabled).

Aufgabe 37:

Implementiert werden soll in dieser Aufgabe das sogenannte Münzen-Spiel. Münzen ist ein Spiel für N (> 1) Spieler mit den Nummern 1 bis N. Vor den Spielern liegt eine Reihe von M (> 1) Münzen. Die Münzen haben ggfls. unterschiedliche Werte. Die Werte sind sichtbar. Ziel von jedem Spieler ist es, eine höhere Anzahl an Münzwerten zu sammeln als die anderen Spieler. Genommen wird der Reihe nach. Es besteht Zugzwang. Ein Spielzug sieht so aus, dass der Spieler, der an der Reihe ist, entweder die Münze ganz rechts oder die Münze ganz links nimmt. Das Spiel ist beendet, wenn alle Münzen genommen wurden. Sieger sind die Spieler mit der höchsten Anzahl an gesammelten Münzwerten.

Aufgabe: Implementieren Sie ein Java-Programm mit einer GUI, mit dem N menschliche Spieler gegeneinander das Münzen spielen können.

Die Parameter des Spiels können dabei im Sourcecode als Konstanten fest vorgegeben werden. Ihr Programm muss prinzipiell aber auch mit einer anderen Konfiguration funktionieren:

```
final public static int ANZAHL_MUENZEN = 12; // groesser als 1
final public static int ANZAHL_SPIELER = 3;  // groesser als 1
final public static int MIN_MUENZWERT = 1;   // groesser als 0
final public static int MAX_MUENZWERT = 6;
                                    // groesser als MIN_MUENZWERT
```

Die GUI des Spiels soll exakt folgende Gestalt haben:

Im oberen Bereich des Fensters werden die einzelnen Münzen des Spiels durch Buttons mit den entsprechenden Münzwerten repräsentiert. Unten im Fenster befindet sich links ein Label für Textausgaben und rechts befindet sich für jeden Spieler ein Label mit dem aktuellen Punktestand des Spielers

Gespielt wird gemäß der obigen Regeln:

- Nur diejenigen Buttons sind jeweils anklickbar (setEnabled) und mit grünem Hintergrund hinterlegt (setBackground), die aktuell genommen werden können, d.h. die Spieler können keine ungültigen Spielzüge tätigen.
- Der aktuelle Spieler kann einen Spielzug ausführen, indem er auf einen aktivierten Button klickt. Der Spielzug wird gemäß den Regeln ausgeführt. Der Button wird geleert.
- Im Label links erscheint die Meldung, welcher Spieler als nächstes an der Reihe ist. Bei Spielende erscheint dort die Meldung „Spielende!“.
- Die Spieler-Label geben für jeden Spieler jeweils die aktuelle Punktezahl des Spielers aus. Bei Spielende geben die Spieler-Label der Spieler, die gewonnen haben, zusätzlich aus: „Sieger!“.

Aufgabe 38:

Gegeben sei die folgende Klasse **`Mensch`** und das folgende Interface **`Bedingung`**:

```
class Mensch {
     private int alter;

     public Mensch(int alter) {    this.alter = alter;     }

     public int getAlter() { return this.alter;      }

     public void setAlter(int alter) {   this.alter = alter;     }
}

interface Bedingung {
     boolean istErfuellt(Mensch mensch);
}
```

Teilaufgabe 1:

Definieren Sie zunächst eine Klasse **`Volljaehrig`**, die das Interface **`Bedingung`** implementiert. Die Methode **`istErfuellt`** soll genau dann *true* liefern, wenn der als Parameter übergebene Mensch älter oder gleich 18 Jahre alt ist.

Teilaufgabe 2:

Definieren Sie anschließend eine Klasse **`MenschArrayList`** als Ableitung der Klasse **`java.util.ArrayList<Mensch>`**. Die Klasse soll eine Methode mit folgender Signatur implementieren:

```
public MenschArrayList filtern(Bedingung bedingung)
```

Die Methode **`filtern`** erzeugt, füllt und liefert dabei ein neu erzeugtes Objekt der Klasse **`MenschArrayList`**, das genau die **`Mensch`**-Objekte enthält, die aktuell in dem aufgerufenen **`MenschArrayList`**-Objekt gespeichert sind und die übergebene Bedingung erfüllen.

Teilaufgabe 3:

Implementieren Sie anschließend ein Testprogramm, das folgendes macht:

1. Es wird ein Objekt der Klasse **`MenschArrayList`** erzeugt.
2. In einer Schleife werden anschließend 10 **`Mensch`**-Objekte mit einem zufällig ermittelten Alter zwischen 1 und 80 Jahren erzeugt und der MenschArrayList aus Schritt 1 hinzugefügt.
3. Durch Aufruf der Methode **`filtern`** werden die volljährigen Menschen der MenschArrayList aus Schritt 1 ermittelt.
4. In einer Schleife wird das jeweilige Alter der in Schritt 3 ausgefilterten volljährigen Menschen auf die Konsole ausgegeben.

Aufgabe 39:

In der JDK-Klassenbibliothek gibt es folgende zwei Interfaces:

```
/**
 * Implementing this interface allows an object to be the target of
 * the "for-each loop" statement.
 * @param <T> the type of elements returned by the iterator
 */
public interface Iterable<T> {
    /**
     * Returns an iterator over elements of type {@code T}.
     * @return an Iterator.
     */
    Iterator<T> iterator();
}

/**
 * An iterator over a collection.
 * @param <E> the type of elements returned by this iterator
 */
public interface Iterator<E> {
    /**
     * Returns {@code true} if the iteration has more elements.
     * (In other words, returns {@code true} if {@link #next} would
     * return an element rather than throwing an exception.)
     *
     * @return {@code true} if the iteration has more elements
```

```
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration
     *  @throws NoSuchElementException if the iteration has no more
elements
     */
    E next();
}
```

Ein Iterator erlaubt also, alle in einer Collection (z.B. eine ArrayList) gespeicherten Elemente nacheinander abzufragen. Eine Klasse, die das Interface `Iterable` implementiert, kann übrigens auch mit der for-each-Schleife benutzt werden!

Sie besitzen den Quellcode der aus der Vorlesung bekannten Klasse `List`, die eine verkettete Liste implementiert:

```
class Element {
  int value; // Speicher fuer einen Wert
  Element next; // Referenz auf das folgende Element (null fuer Ende)

  Element(int v, Element n) {
    this.value = v;
    this.next = n;
  }
}


class List {
  Element first; // erstes Element der Liste
  Element last;  // letztes Element der Liste

  List() {
    this.first = null;
    this.last = null;
  }

  void append(int i) {
    Element elem = new Element(i, null);
    if (this.first == null) { // erstes Element in der Liste
      this.first = elem;
      this.last = elem;
    } else { // Anhaengen des Elementes am Ende
      this.last.next = elem;
      this.last = elem;
    }
  }
}
```

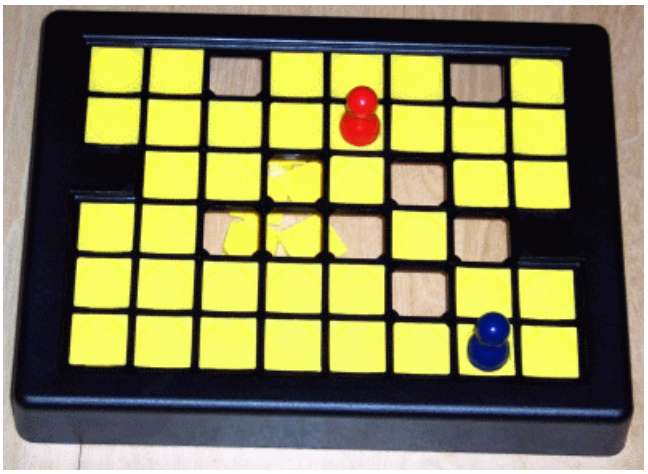
Aufgabe: Ändern bzw. ergänzen Sie den Quellcode so, dass die Klasse `List` das Interface `Iterable` adäquat implementiert. Folgendes Beispielprogramm würde bei korrekter Implementierung der Klasse `List` zweimal folgende Ausgabe erzeugen:

```
8 -> 6 -> 2 -> 3 -> 7 -> 1 -> 2 ->

  public static void main(String[] args) {
```

```
    List list = new List();
    list.append(8);
    list.append(6);
    list.append(2);
    list.append(3);
    list.append(7);
    list.append(1);
    list.append(2);

    Iterator<Integer> it = list.iterator();
    while (it.hasNext()) { // direkte Nutzung des Iterators
      System.out.print(it.next() + " -> ");
    }
    System.out.println();

    for (Integer i : list) { // Nutzung mit for-each-Schleife
      System.out.print(i + " -> ");
    }
  }
}
```

Aufgabe 40:

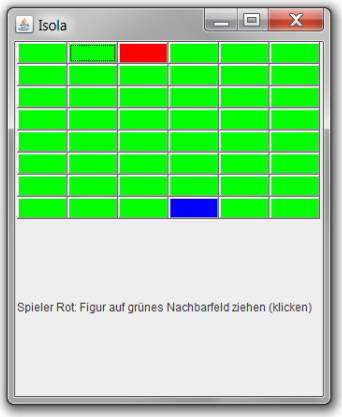
Implementiert werden soll in dieser Aufgabe das sogenannte Isola-Spiel.



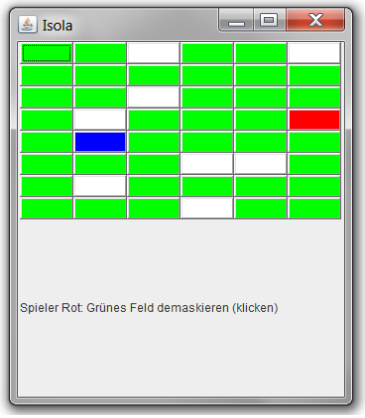
Aus Wikipedia: Isola ist ein strategisches Brettspiel für zwei Spieler. Es wurde 1972 und 1980 vom Ravensburger-Verlag herausgegeben. Es wird auf einem 8×6 Felder großen Spielfeld gespielt. Jeder Spieler hat eine Spielfigur, außerdem benötigt man noch 46 Plättchen, um die erlaubten Felder zu markieren. Zur Vorbereitung werden die beiden Spielfiguren auf die Felder A3 und H4 gestellt, hier liegen keine Plättchen. Alle anderen Felder werden mit den Plättchen belegt. Ein Spielzug besteht aus zwei Schritten: Zuerst bewegt der Spieler seine Spielfigur auf eines der (maximal acht) umliegenden Plättchen, wobei er nicht auf das Feld mit dem Gegner und nicht auf ein leeres Feld ziehen darf. Dann entfernt er ein beliebiges Plättchen, das von keinem der beiden Spieler belegt ist. Wer zuerst nicht mehr ziehen kann, hat das Spiel verloren.

Aufgabe: Implementieren Sie ein Java-Programm mit einer GUI, mit dem zwei menschliche Spieler gegeneinander das Münzen spielen können.

Die GUI des Spiels soll exakt folgende Gestalt haben:



Anfang



Während Spiel

Ende

Im oberen Bereich des Fensters wird das Spielbrett durch 8 * 6 Buttons repräsentiert. Unten im Fenster befindet sich ein Label für Textausgaben. Der Button, auf dem sich aktuell die Spielfigur von Spieler A befindet, wird rot dargestellt. Der Button, auf dem sich aktuell die Spielfigur von Spieler B befindet, wird blau dargestellt. Felder mit Plättchen werden grün dargestellt. Felder ohne Plättchen werden weiß dargestellt.

Gespielt wird gemäß der obigen Regeln:

- Die Spieler ziehen abwechselnd.
- Jeder Spielzug besteht aus zwei Phasen.
- In der ersten Phase zieht der aktuelle Spieler seine Figur auf ein Nachbarfeld (Anklicken des entsprechenden Buttons)
- In der zweiten Phase entfernt der aktuelle Spieler ein Plättchen, d.h. demaskiert das entsprechende Feld (Anklicken des entsprechenden Buttons)

Beachten Sie:

- Halten Sie sich, was die Textausgaben im Label angeht, an die Ausgaben meines Demo-Programms.
- Es sollen jeweils nur die Buttons anklickbar sein (enabled!), die auch gemäß der Spielregeln ausgewählt werden dürfen. D.h. ein Spieler kann keine falschen Spielzüge machen.
- Bei Spielende sollen alle Buttons disabled werden.

Aufgabe 41:

Gegeben sei das folgende Interface **`Container`**, das vier Methoden für so genannte Container-Klassen definiert. Container-Klassen sind ADT-Klassen zur Speicherung von Daten, die jeweils eine spezielle Semantik verfolgen.

```
interface Container {
  public void put(int value); // einfuegen
  public int get(); // rausholen
  public boolean isEmpty(); // ist leer?
  public boolean isFull(); // ist voll?
}
```

Ein spezieller Container-Typ sind die aus der Vorlesung bekannten Stacks. Ein Programmierer hat folgende Klasse **`Stack`** implementiert, die das Interface implementiert.

```
class Stack implements Container {

  private ArrayList<Integer> values = new ArrayList<Integer>();

  final public void put(int value) {  values.add(value); }
  public int get() {  return values.remove(values.size() - 1); }
  final public boolean isEmpty() {  return values.size() == 0; }
  final public boolean isFull() {  return false; }
}
```

Durch die Methode **`put`** wird also ein Wert dem Stack hinzugefügt, durch die Methode **`get`** wird der jüngste, d.h. als letztes hinzugefügte Wert vom Stack entfernt und geliefert.

Einen anderen Container-Typ stellen Queues dar. Queues verfolgen die Semantik, dass analog zu Stacks mittels **`put`** ein Wert der Queue hinzugefügt werden kann. Mittels **`get`** wird aber nicht der jüngste, sondern der älteste Wert, d.h. der Wert, der jeweils am längsten in der Queue gespeichert ist, aus der Queue entfernt und geliefert.

Der Unterschied zwischen Queues und Stacks liegt also lediglich an einer anderen Implementierung der Methode **`get`**. Eine Klasse **`Queue`**, die das Interface **`Container`** implementiert, kann also einfach dadurch definiert werden, dass sie von der Klasse **`Stack`** abgeleitet wird und lediglich die Methode **`get`** adäquat überschreibt.

Aufgabe: Definieren Sie eine Klasse **`Queue`**, die Queues entsprechend der oben beschriebenen Semantik umsetzt. Leiten Sie die Klasse **`Queue`** dabei von der obigen Klasse **`Stack`** ab und überschreiben Sie die Methode **`get`**. Beachten Sie, dass die interne Datenstruktur (**`java.util.ArrayList`**) als **private** deklariert ist und alle Methoden außer **`get`** als **`final`** deklariert sind. Das Interface **`Container`** und die Klasse **`Stack`** dürfen nicht verändert werden.

Tipp: Nutzen Sie in der Methode **`get`** einen Hilfsstack und die geerbten Methoden **`put`** und **`get`**, um an das älteste Element zu gelangen.

Aufgabe 42:

In einem Abenteuerspiel gibt es Spielfiguren. Jede davon kann Drohungen ausstoßen. Eine konkrete Spielfigur ist beispielsweise ein Monster.

```
interface Spielfigur {
     void drohe();
}

class Monster implements Spielfigur {

     public void drohe() {
           System.out.print("Grrr!");
     }
```

```
}
```

Alle Spielfiguren können sich zur Laufzeit Krankheiten, wie Schnupfen bzw. Husten einfangen – dann geht ihren Drohungen bspw. ein Schniefen bzw. Husten voraus. Theoretisch ließe sich das über entsprechende Unterklassen von bspw. **`Monster`** wie **`VerschnupftesMonster`** oder **`VerhustetesVerschnupftesMonster`** oder **`VerschnupftesVerhustetesMonster`** umsetzen.

```
class VerschnupftesMonster extends Monster {

     public void drohe() {
           System.out.print("Schnifff! Grrr!");
     }
}
```

Dieser Ansatz führt jedoch bei Hinzunahme weiterer Krankheiten wie „Heiserkeit“ oder „Zahnschmerzen“ zu einer exponentiellen Zunahme der Unterklassen, ganz abgesehen von unerträglich langen Klassennamen.

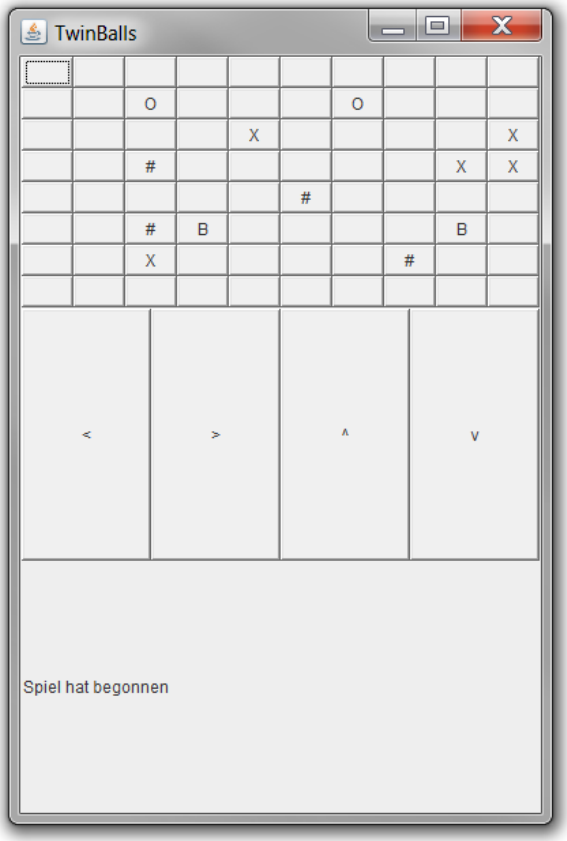
Eine flexible und erweiterbare Alternative bildet der Einsatz des sogenannten Dekorator-Musters durch den Aufbau von Pipeline-ähnlichen Strukturen:

```
Spielfigur meinMonster = new Monster();
meinMonster.drohe(); // Grrr!

Spielfigur meinVerhustetesMonster = new Husten(meinMonster);
meinVerhustetesMonster.drohe(); // Hust! Grrr!


Spielfigur meinVerschnupftesMonster = new Schnupfen(meinMonster);
meinVerschnupftesMonster.drohe(); // Schnifff! Grrr!

Spielfigur meinVerschnupftesVerhustetesMonster =
  new Schnupfen(new Husten(meinMonster));
meinVerschnupftesVerhustetesMonster.drohe();
// Schnifff! Hust! Grrr!

Spielfigur meinVerhustetesVerschnupftesMonster =
  new Husten(new Schnupfen(meinMonster));
meinVerhustetesVerschnupftesMonster.drohe();
// Hust! Schnifff! Grrr!

Spielfigur meinHeiseresVerschnupftesMonster =
  new Heiserkeit(new Schnupfen(meinMonster));
meinHeiseresVerschnupftesMonster.drohe();
// Röchel! Schnifff! Grrr!
```

Aufgabe: Überlegen Sie aufbauend auf den Konzepten der Polymorphie und des dynamischen Bindens ein Konzept zur Realisierung obiger Pipelines für beliebige Krankheiten von beliebigen Spielfiguren! Implementieren Sie konkret die Krankheit-Klassen **`Husten`**, **`Schnupfen`** und **`Heiserkeit`**, so dass sich das Beispielprogramm übersetzen lässt, die gewünschte Pipeline realisiert und die in den Kommentaren angedeuteten Ausgaben erzeugt!

Aufgabe 43:

Implementiert werden soll in dieser Aufgabe das sogenannte TwinBalls-Spiel.



TwinBalls ist ein Knobelspiel für eine Person. Das eigentliche Spielfeld besteht aus n*m Feldern, die entweder leer (entspricht Leerzeichen) oder mit einem der folgenden Zeichen markiert sind:

```
B = Ball
O = Ziel
# = Mauer
X = Loch
```

Dabei gibt es jeweils genau zwei Bälle. Ziel des Spiels ist es, beide Bälle gleichzeitig auf Ziele zu bewegen.

In jedem Spielzug entscheidet sich der Spieler für eine der vier Aktionen links, rechts, hoch oder runter, um beide Bälle gleichzeitig in die entsprechende Richtung zu bewegen. Nach Auswahl der Aktion wird vor dem Bewegen der Bälle allerdings zunächst überprüft, ob der Spielzug gültig ist. Ein Spielzug ist dabei ungültig und das Spiel unmittelbar verloren, wenn

- einer der beiden oder beide Bälle das Spielfeld verlassen würde,
- einer der beiden oder beide Bälle in ein Loch fallen würde,
- einer der beiden aber nicht beide Bälle ein Ziel erreichen würde.

Ist der Spielzug gültig, werden beide Bälle gleichzeitig um ein Feld in die entsprechende Richtung bewegt, wobei jedoch eine Mauer den Ball blockiert und er auf dem alten Feld bleibt. Erreichen beide Bälle ein Ziel, hat der Spieler das Spiel gewonnen.

Aufgabe: Implementieren Sie das TwinBalls-Spiel in Form einer Java-GUI-Anwendung. Die Darstellung erfolgt dabei wie in der Abbildung angedeutet. Im oberen Bereich des Fensters wird das Spielfeld dargestellt, wobei jedes Feld durch einen Button mit dem entsprechenden Zeichen als Label repräsentiert wird. Unterhalb des Spielfeldes befinden sich vier Aktions-Buttons, die die vier Aktionen links, rechts, hoch und runter repräsentieren. Ganz unten im Fenster wird ein Label für die Ausgabe von Nachrichten platziert.

Das Ausgangsspielfeld befindet sich in einer Datei namens *feld.txt* und kann mit folgendem Befehl eingelesen werden:

```
char[][] spielfeld = IO.readFileAsCharMatrix("feld.txt");
```

Sie können davon ausgehen, dass die Datei nur gültige Spielfelder repräsentiert.

Nach Start des Programms kann der Spieler jeweils die vier Aktionsbuttons anklicken, wobei der entsprechende Spielzug überprüft und falls gültig anschließend ausgeführt wird.

Bei einem ungültigen Spielzug wird das Spiel sofort beendet und im Nachrichten-Label „Leider verloren!“ ausgegeben. Hat der Spieler gewonnen, wird das Spiel ebenfalls sofort beendet und im Nachrichten-Label der Text „Gewonnen!“ ausgegeben. Nach Spielende wird beim Anklicken einer der vier Aktionsbuttons keine Aktion mehr ausgeführt.

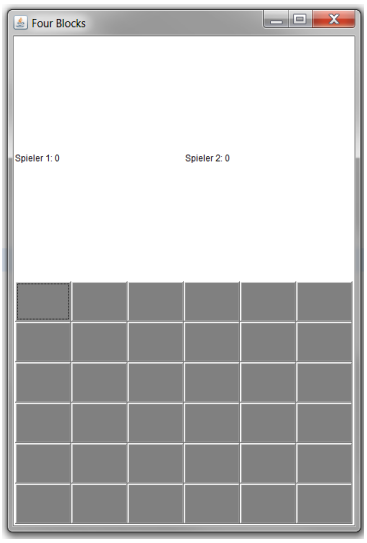
Aufgabe 44:

Implementiert werden soll in dieser Aufgabe das sogenannte FourBlocks-Spiel.

FourBlocks ist ein Spiel für 2 Spieler. Gespielt wird auf einem Spielbrett mit N * N Feldern (N > 3). Anfangs sind alle Felder leer. Spieler 1 besitzt grüne Steine, Spieler 2 gelbe Steine. Gespielt wird abwechselnd, Spieler 1 beginnt. Ein Spielzug besteht daraus, dass der Spieler, der an der Reihe ist, auf einem leeren Feld einen Spielstein seiner Farbe platziert. Für jeden quadratischen 4er-Block mit Spielsteinen seiner Farbe, der durch das Setzen eines Steines gebildet wird, erhält er einen Punkt (bis zu 4 Punkte pro Spielzug sind möglich). Das Spiel endet, wenn alle Felder belegt sind. Gewonnen hat der Spieler mit den meisten Punkten. Haben beide Spieler gleich viele Punkte, endet das Spiel unentschieden.

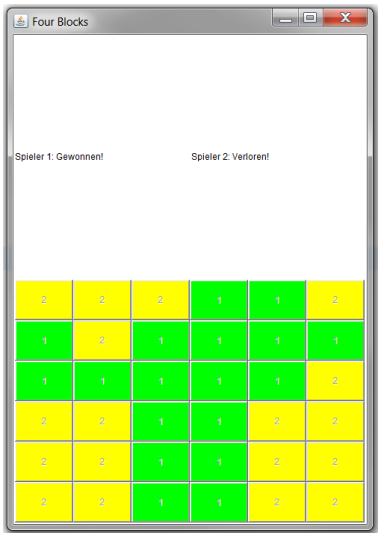
Aufgabe: Implementieren Sie ein Java-Programm mit einer GUI, mit dem zwei menschliche Spieler gegeneinander das Spiel FourBlocks spielen können. Nutzen Sie für die Größe des Feldes N eine Konstante.

Die GUI des Spiels soll exakt folgende Gestalt haben:



Anfang

Während Spiel



Ende

Oben im Fenster befinden sich zwei Label für die Ausgabe der Punktestände der beiden Spieler. Im unteren Bereich des Fensters wird das Spielbrett durch N*N Buttons repräsentiert. Leere Felder werden durch graue Buttons repräsentiert. Steine von Spieler 1 werden durch grüne Buttons mit der Beschriftung 1 repräsentiert. Steine von Spieler 2 werden durch gelbe Buttons mit der Beschriftung 2 repräsentiert.

Gespielt wird gemäß der obigen Regeln:

- Die Spieler ziehen abwechselnd.
- Ein Spielzug besteht daraus, dass der Spieler, der an der Reihe ist, ein leeres Feld (grauer Button) anklickt.
- Der entsprechende Button wird in der Spielerfarbe eingefärbt und mit der entsprechenden Beschriftung (1 oder 2) versehen. Außerdem wird er disabled, so dass er nicht mehr anklickbar ist.
- Es wird überprüft, ob Vierer-Blöcke entstanden sind. Falls ja, wird der Spielstand des Spielers entsprechend erhöht und die Ausgabe im entsprechenden Spieler-Label angepasst.
- Bei Spielende wird in den Spieler-Labeln ausgegeben „Gewonnen“, „Verloren“ oder „Unentschieden“.

Aufgabe 45:

2048 ist ein Computerspiel, das von einem einzelnen Spieler gespielt werden kann. Ziel des Spiels ist das Erstellen einer Kachel mit der Zahl 2048 durch das Verschieben und Kombinieren anderer Kacheln (Quelle: Wikipedia)

2048 wird auf einem Spielfeld mit N×N Kästchen gespielt (hier Konstante N = 4, aber prinzipiell auch höhere Werte möglich!), auf dem sich Kacheln befinden, die mit Zweierpotenzen (2, 4, 8, 16, …) beschriftet sind.

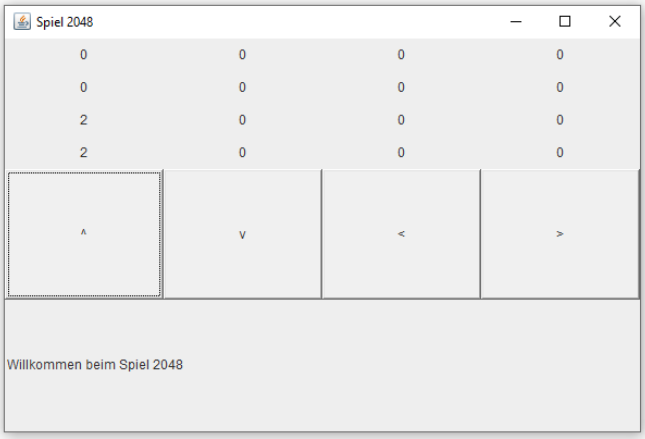
Am Anfang befinden sich auf dem Spielfeld zwei zufällige Kacheln, die jeweils eine 2 tragen. Mit Pfeiltasten (oben, unten, links, rechts) bewegt der Spieler die Kacheln auf dem Spielfeld, wobei sich bei jedem Zug alle Kacheln so weit wie möglich bewegen, als ob sie auf dem in die jeweilige Richtung gekippten Spielfeld rutschen würden. Stoßen dabei zwei Kacheln mit der gleichen Zahl aneinander, verschmelzen sie zu einer Kachel mit der Summe der beiden Kacheln. Zusätzlich entsteht mit jedem Zug, aber nur falls es im vorherigen Zug eine Verschiebung gegeben hat, in einem leeren Feld eine zufällige Kachel mit einer 2.

Das Ziel des Spiels ist das Bilden einer Kachel mit der Zahl 2048. Dann hat der Spieler das Spiel gewonnen. Das Spiel endet auch, wenn alle Felder mit Kacheln belegt sind und der Spieler keinen Zug mehr machen kann. Dann hat der Spieler verloren.

Teilaufgabe (a):

Implementieren Sie mit dem Java-AWT-GUI-Framework exakt folgende GUI für das Spiel 2048:

Oben im Fenster befindet sich ein Panel mit N*N Labeln (hier Konstante N = 4, aber prinzipiell auch höher). Die Label repräsentieren die einzelnen Kacheln mit den jeweiligen Zahlen. Leere Kacheln enthalten eine 0. Darunter befindet ein Panel mit 4 Buttons für die Steuerung (oben, unten, links, rechts). Ganz unten befindet sich ein Label für Nachrichten.



Teilaufgabe (b):

Machen Sie die GUI interaktiv, so dass man das Spiel 2048 spielen kann:

- Anfangs liegt auf zwei zufällig gewählten Kacheln jeweils eine 2. Im Nachrichten-Label steht „Willkommen beim Spiel 2048“.
- Klickt der Benutzer auf einen der vier Steuerungsbuttons, verrutschen die Kacheln (genauer gesagt, die Beschriftungen der Label), wie in der Spielanleitung beschrieben, in die entsprechende Richtung. Bei den Richtungen „oben“ und „links“ werden die Kachel dabei in der Reihenfolge von oben nach unten und jeweils von links nach rechts betrachtet. Bei den

Richtungen „unten“ und „rechts“ werden die Kachel in der Reihenfolge von unten nach oben und jeweils von rechts nach links betrachtet.

- Nach dem Verrutschen (und auch nur, wenn mindestens eine Kachel tatsächlich verrutscht ist), wird auf einer zufällig ausgewählten leeren Kachel eine 2 gesetzt.
- Gewinnt der Spieler, werden die vier Steuerungsbuttons disabled und es erfolgt im Nachrichtenlabel die Ausgabe „Gewonnen! Herzlichen Glückwunsch!“.
- Verliert der Spieler, werden die vier Steuerungsbuttons disabled und es erfolgt im Nachrichtenlabel die Ausgabe „Leider verloren!“

.